Sven Sonnenburg

# Entwicklung eines modularen Softwaresystems auf der Basis von OSGi-Komponenten prototypisch am Beispiel eines Shop-Systems

BACHELORARBEIT

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCE

Fakultät Mathematik/Naturwissenschaften/Informatik

Jena, 2011

Sven Sonnenburg

# Entwicklung eines modularen Softwaresystems auf der Basis von OSGi-Komponenten prototypisch am Beispiel eines Shop-Systems

eingereicht als

BACHELORARBEIT

an der

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCE

Fakultät Mathematik/Naturwissenschaften/Informatik

Jena, 2011

Erstprüfer: Prof. Dr.-Ing. Mario Geissler

Zweitprüfer: Dipl.-Ing. Jens Rössel

Vorgelegte Arbeit wurde verteidigt am:

**Bibliographische Beschreibung:**

Sonnenburg, Sven:
Entwicklung eines modularen Softwaresystems auf der Basis von OSGi-Komponenten prototypisch am Beispiel eines Shop-Systems. – 2011. – 56 S.
Mittweida, Hochschule Mittweida (FH), Fachbereich Mathematik/Naturwissenschaften/ Informatik, Bachelorarbeit, 2011

Referat:

Diese Bachelorarbeit beschäftigt sich mit der Entwicklung eines modularen Softwaresystems. Dies soll anhand eines Shop-Systems umgesetzt werden. Ziel ist es einen lauffähigen Prototypen zu erstellen. Dieser soll aus Modulen bestehen und damit das Pluggable-Konzept umsetzen.

Zuerst werden wichtige Begriffe erläutert um einen theoretischen Einstieg zu gewährleisten. Aufbauend auf dieser Basis werden verschiedene bekannte Shop-Systeme im Open Source Bereich ausgewählt und deren Aufbau mit einander verglichen.

Im nächsten Punkt wird anhand der gewonnen Erkenntnisse und der vorhanden Anforderungen ein Grundkonzept für den Prototyp erstellt. Und mögliche Ansätze besprochen. Daraufhin erfolgt eine Auswahl von weiteren benötigten Technologien, um das erstellte Grundkonzept umsetzen zu können. Nachfolgend wird die eigentliche Implementierung erläutert und eventuell auftretende Probleme aufgeführt.

Im letzten Schritt werden die gewonnen Erkenntnisse aus den vorherigen Kapiteln zusammengefasst und ausgewertet. Darauf aufbauend erfolgt ein genereller Ausblick.

# Inhaltsverzeichnis

# Abbildungsverzeichnis



# Listings



# Tabellenverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| **AJAX** | Asynchronous JavaScript and XML |
| **CSS** | Cascading Style Sheets |
| **e-Commerce** | electronic Commerce |
| **E-Shop** | Electronic Shop |
| **GUI** | Graphical User Interface |
| **GWT** | Google Web Toolkit |
| **HTML** | HyperText Markup Language |
| **IMAP** | Internet Message Access Protocol |
| **JAR** | Java Archive |
| **JVM** | Java Virtual Machine |
| **OSGi** | Open Services Gateway initiative |
| **PHP** | PHP: Hypertext Preprocessor |
| **POP3** | Post Office Protocol (Version 3) |
| **RPC** | Remote-Procedure-Call |
| **SLF4J** | Simple Logging Facade for Java |
| **SMTP** | Simple Mail Transfer Protocol |
| **Spring DM** | Spring Dynamic Modules |
| **SQL** | Structured Query Language |
| **WAR** | Web Application Archive |
| **XML** | Extensible Markup Language |

# 0 Einleitung

## 0.1 Motivation

In unserer heutigen Zeit ist das Internet für viele Menschen zu einem wichtigen Bestandteil ihres Lebens geworden. Man sucht nach Informationen, liest Nachrichten, kommuniziert mit anderen Menschen über soziale Netzwerke und Instant Messenger. Man spielt mit anderen zusammen oder gegeneinander Online Spiele. Wir kaufen sogar über das Internet ein.

Wenn man vom Einkaufen im Internet spricht, umgangssprachlich Onlineshopping, betreten wir den Bereich des e-Commerce. Es geht also um den elektronischen Handel im virtuellen Netz. Mittlerweile gibt es eine große Anzahl von Unternehmen, die ihre Produkte im Internet vertreiben. So vielfältig die angebotenen Produkte sind, so vielfältig sind auch die Anforderungen an einen Online-Shop. Und damit ist nicht nur die grafische Präsentation im Internet gemeint. Zum Beispiel hat ein Shop, der Kleidung verkauft, andere Anforderungen zu erfüllen als einer, der Baustoffe vertreibt. Um Kleidung einordnen zu können, braucht man Attribute wie Größe, Farbe, Schnitt, Angabe Preis pro Stück, Rabatt usw. Für Baustoffe benötigt man wiederum andere Attribute wie Größe, Gewicht, Preis pro Gewichtseinheit. Und was wäre, wenn jemand zwei völlig verschiedene Produkte in einem Shop anbieten will? Wie sieht es überhaupt aus wenn man seine Produkte in mehreren Ländern vertreiben möchte? Das Stichwort lautet Internationalisierung, also zum Beispiel Mehrsprachigkeit und die damit verbundenen unterschiedlichen Währungen, Größen und Einheiten. All diese verschiedenen Anforderungen betreffen viele, wenn nicht sogar alle Elemente eines Shop-Systems. Angefangen mit der Präsentation, den Schnittstellen, der Geschäftslogik, die Datenhaltung um nur einiges zu nennen.

Damit stellt sich immer mehr die Frage, ob es so ein Shop-System überhaupt gibt, das alle diese verschiedenen Anforderungen abdecken und erfüllen kann. Und wenn ja, wie müsste so ein Alleskönner denn aussehen? Welche Eigenschaften müsste er haben? Wie müsste er aufgebaut sein? Gibt es überhaupt schon eine oder mehrere Technologien, mit dem man so etwas bewerkstelligen kann?

## 0.2 Aufgabenstellung und Ziele

Ziel der Bachelorarbeit ist es, erst einmal festzustellen was modulare Softwaresysteme überhaupt sind. Diese Thematik soll dann auf das spezielle Beispiel Shop-Systeme übertragen werden.

Danach sollen existierende Shop-Systeme anhand definierter Kriterien näher betrachtet werden. Darauf aufbauend soll eine selbst ausgewählte Alternative gegenübergestellt werden.

Ausgehend vom Ergebnis der Gegenüberstellung soll ein Prototyp entwickelt werden, der einem modularen Software-System genügt und gleichzeitig das Pluggable-Konzept umsetzt. Das heißt auch, es muss ein Konzept erstellt werden, welches den Aufbau und die Struktur widerspiegelt. Zudem soll erarbeitet werden, welche Bestandteile nötig sind, um den Prototypen umzusetzen.

Danach sollen mögliche Testszenarien und Testmethoden aufgezeigt werden. Zuletzt soll der entwickelte Prototyp mit den Vorgaben verglichen werden und ein möglicher Ausblick gegeben werden.

## 0.3 Thematische Abgrenzung

Aufgrund des Umfangs der Thematik „Entwicklung eines Shop-Systems“ wird eine thematische Abgrenzung vorgenommen.

Es soll ein Kernmodul entwickelt werden, das prinzipielle Basisfunktionalitäten für das Shop-System beinhaltet. Diese Funktionalitäten sind zu erarbeiten. Für die Anzeige wird ein Modul benötigt, das zur Präsentation der Daten dient. Eine GUI, bzw. in diesem Fall ein Front-End. Außerdem wird eine zweite GUI benötigt, ein Back-End. Dies Modul soll zur administrativen Verwaltung des Shop-Systems dienen. Daher ist es erst einmal ausreichend, eine einfache Bestandsverwaltung zu haben, d.h. eine einfache Kategorieverwaltung und Produktverwaltung zu implementieren.

Des Weiteren ist es ausreichend, für den Prototypen eine 2-Level-Kategorie zu verwenden, sowie Simple-Articles. Damit sind zum Beispiel Produktvariationen ausgeschlossen. Es wird außerdem noch ein einfaches Check-out-Modul benötigt. Bei der Entwicklung der grafischen

Komponenten wird gänzlich auf Cross-Browser-Kompatibilität verzichtet. Das heißt es wird explizit nur der Internetbrowser Mozilla Firefox in den Version 4.0 und 5.0 unterstützt.

Zudem muss noch eine passende Möglichkeit der Datenhaltung gefunden werden. Weiter Rahmenbindungen sind, dass das Shop-System erst einmal nur für den deutschen Markt ausgelegt sein muss. Zusätzlich werden die rechtlichen Gesichtspunkte eines Shop-Systems größtenteils außer Acht gelassen.

Wie schon in den vorherigen Punkten erwähnt, ist darauf zu achten, das Pluggable-Konzept bei der Entwicklung zu berücksichtigen.

## 0.4 Aufbau der Arbeit

Im ersten Punkt dieser Bachelorarbeit werden wichtige theoretische Grundlagen erläutert. Dies soll zum Einstieg in die Arbeit dienen. Und ist gleichzeitig die theoretische Grundlage für die noch folgenden Kapitel.

Darauffolgend im zweiten Kapitel, werden im ersten Schritt gängige Shop-Systeme ausgewählt. Die Auswahl wird auf externe Vergleiche und Ranglisten gestützt. Zu dem spielen die verwendeten Technologien eine wichtige Rolle, um ein größeres Spektrum abdecken zu können. Im nächsten Schritt wird der generelle Aufbau untersucht. Das heißt es soll festgestellt werden, ob die ausgewählten Shop-Systeme den Ansprüchen eines modularen Softwaresystems genügen. Zusätzlich wird Wert auf den Aspekt der Erweiterbarkeit und die Umsetzung des Pluggable-Konzeptes gelegt. An diesem Punkt können vielleicht schon Schwachstellen der ausgewählten Shop-Systeme herausgefiltert werden. Anschließend wird im nachfolgenden Kapitel eine geeignete Basistechnologie für den Prototyp des Shop-Systems ermittelt.

Im vierten Kapitel wird zuerst ein grobes Konzept für den Prototyp entwickelt und erstellt. Auf Grundlage des Grob-Konzeptes für den Prototyp werden benötigte Technologien und Elemente herausgefiltert. Auch diese Bestandteile werden einer Analyse unterzogen und die beste Variante ausgewählt. Nachfolgend wird ein detaillierteres Prototyp-Konzept entwickelt, welches auf dem Grob-Konzept und auf den zuvor ausgewählten Technologien basiert.

Als letzter Schritt in diesem Kapitel folgt die Implementierung des Prototyps mit den ausgewählten Technologien und Elementen anhand der Konzeption. Zusätzlich werden einzelne Module näher erläutert. Anhand von Programmcode Beispielen wird auch die

Umsetzung des Pluggable-Konzeptes näher beleuchtet. Außerdem werden eventuelle Schwierigkeiten und Hindernisse bei der Implementierung besprochen.

Das fünfte Kapitel beschäftigt sich mit dem Thema Softwaretests. Es sollen mögliche und geeignete Methoden, Programme und Technologien gefunden werden, um den entwickelten Prototypen auf Herz und Nieren testen zu können. Zusätzlich sollen noch passende Teststrategien entwickelt werden.

Das letzte Kapitel besteht aus mehreren Teilen. Zum einen soll der entwickelte Prototyp mit den Vorgaben verglichen werden. Zudem wird eine Auswertung vorgenommen. Diese soll möglichst objektiv ausfallen, um auch eventuelle Schwachstellen und Verbesserungen entdecken und herausfiltern zu können. Darauf aufbauend wird ein möglicher Ausblick gegeben.

# 1 Begriffserklärung

In diesem Kapitel werden relevante Begriffe erläutert. Dies dient sowohl zum besseren Verständnis, als auch zum theoretischen Einstieg in die Arbeit. Nachfolgend werden die Begriffe des modularen Softwaresystems, der des Shop-Systems und OSGi näher erklärt.

## 1.1 Modulares Softwaresystem

Ein modulares Softwaresystem muss in kleine Teile zerlegbar sein. Diese Teile werden Module genannt. Ein Modul definiert sich durch folgende Anforderungen. Zu aller erst ist ein Modul unabhängig von anderen Modulen. Dies bedeutet, es muss für sich allein existieren können und selbstständig sein. Diese kleinen Einheiten besitzen außerdem noch eine klar definierte und abgegrenzte Aufgabe. Somit besteht eine klare Aufgabentrennung. Desweiteren definiert sich ein Modul noch dadurch, dass es durch Schnittstellen mit anderen Modulen kommunizieren kann. Das heißt, es muss auf Funktionen anderer Module zugreifen können. Und es kann auch selber eigene Funktionalitäten anbieten. Module besitzen also eine Importschnittstelle und eine Exportschnittstelle. Letztere dient dazu, wie schon erwähnt, Ressourcen nach außen hin zur Verfügung zu stellen, welche wiederum andere Module konsumieren können. Dadurch ist es notwendig, dass die Modulinterna im Modulkörper verborgen bleiben müssen. Somit ergibt sich eine klare Trennung zwischen der Spezifikation des Moduls und dessen eigentlicher Implementierung. Wie schon erwähnt, kommunizieren die einzelnen Komponenten, beziehungsweise die einzelnen Module miteinander. Diese Kommunikation erfolgt durch Schnittstellen. Ein weiterer Punkt ist, dass das Kriterium der Wiederverwendbarkeit erfüllt werden muss.

Wie im oberen Abschnitt beschrieben, besteht ein modulares Softwaresystem aus einzelnen Modulen, die verschiedene Kriterien erfüllen müssen. Damit ähnelt es sehr stark dem Baukastenprinzip. Das heißt man hat eine gewisse Anzahl von Bausteinen, die in bestimmten Abhängigkeiten zueinander stehen. Daher sollte nicht jeder Baustein auf jeden anderen Baustein passen. Dies soll durch die nachfolgende Abbildung 1 einmal veranschaulicht werden.

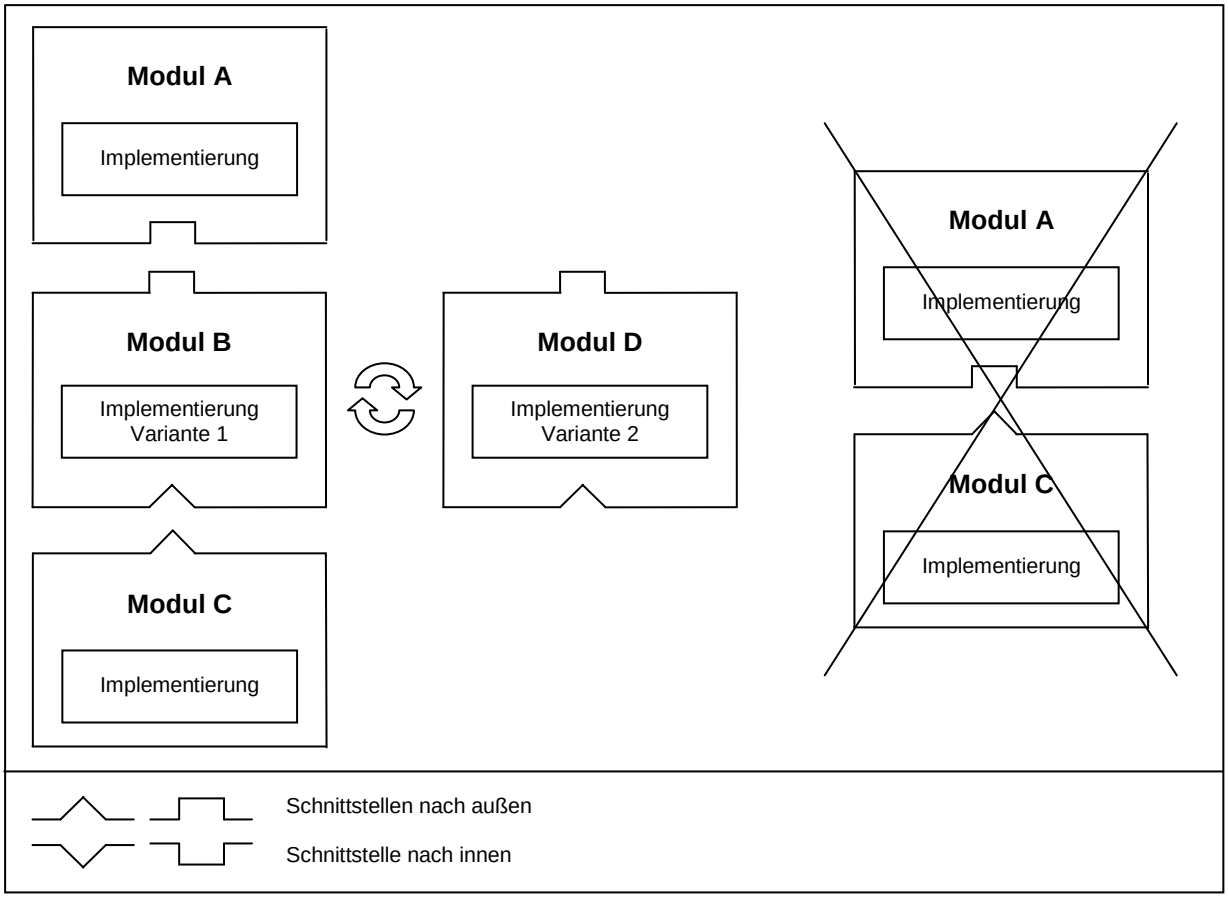



**Abbildung 1: Moduldarstellung**

In der Abbildung 1 sind vier Module dargestellt. Diese definieren jeweils eine oder zwei Schnittstellen nach außen oder innen, welche die Kommunikation zwischen den jeweiligen Modulen A, B, C und D darstellen. Außerdem ist zu erkennen, dass die Implementierung im internen jedes einzelnen Moduls abgegrenzt ist und damit nicht nach außen getragen wird.

Module C stellt eine Ressource, durch eine Schnittstelle nach außen. zur Verfügung. Da Modul B dieselbe Schnittstelle nach innen besitzt, kann es die von C zur Verfügung gestellte Ressource von C nutzen. Dieses Prinzip ist übertragbar auf die Abhängigkeit von Modul B zu Modul A. Was aber, wenn nun Modul A die Ressource von Modul C nutzen will. Dies ist nicht möglich, weil diese beiden Module nicht die gleiche Schnittstelle zur Kommunikation nutzen. Damit ist noch mal festzuhalten. dass die einzelnen Komponenten in gewissen Beziehungen stehen können und nicht unbedingt unter einander kompatibel sind.

Wie steht aber nun Modul B mit Modul D in Verbindung? Wie zu sehen ist, haben beide Module die gleichen Schnittstellen in die richtige Richtung – jeweils die gleiche Schnittstelle nach außen und nach innen. Damit ist Modul D in der Lage Modul B zu ersetzen, ohne dass etwas an Modul C oder A geändert werden muss. Dadurch wurde eine weitere Anforderung,

die an ein Modul gestellt wird, aufgezeigt. Wenn die Schnittstellen, die es benutzt, sowohl nach innen als auch außen gleich bleiben, dann kann ein zweites Modul mit den gleichen Schnittstellen, aber einer anderen Implementierung des ursprünglichen Modules, ersetzen. Diese bedeutet, dass man ein Modul durch ein anderes ersetzen kann, sofern gewisse Rahmenbedingungen eingehalten werden.

Ein weiterer Vorteil dieser modularen Einteilung eines Softwaresystems ist es, einfach ein Modul einzuklinken und an ein vorhandenes Modul anzustecken, ohne das die bisherigen Abhängigkeiten gestört werden. Das Einklinken und Austauschen von Modulen in ein laufendes System, wird im Rahmen der Arbeit, als **Pluggable-Konzept** bezeichnet. Aufbauend auf dem Beispiel in Abbildung 1 wird ein weiteres Modul mit der Bezeichnung E eingefügt.

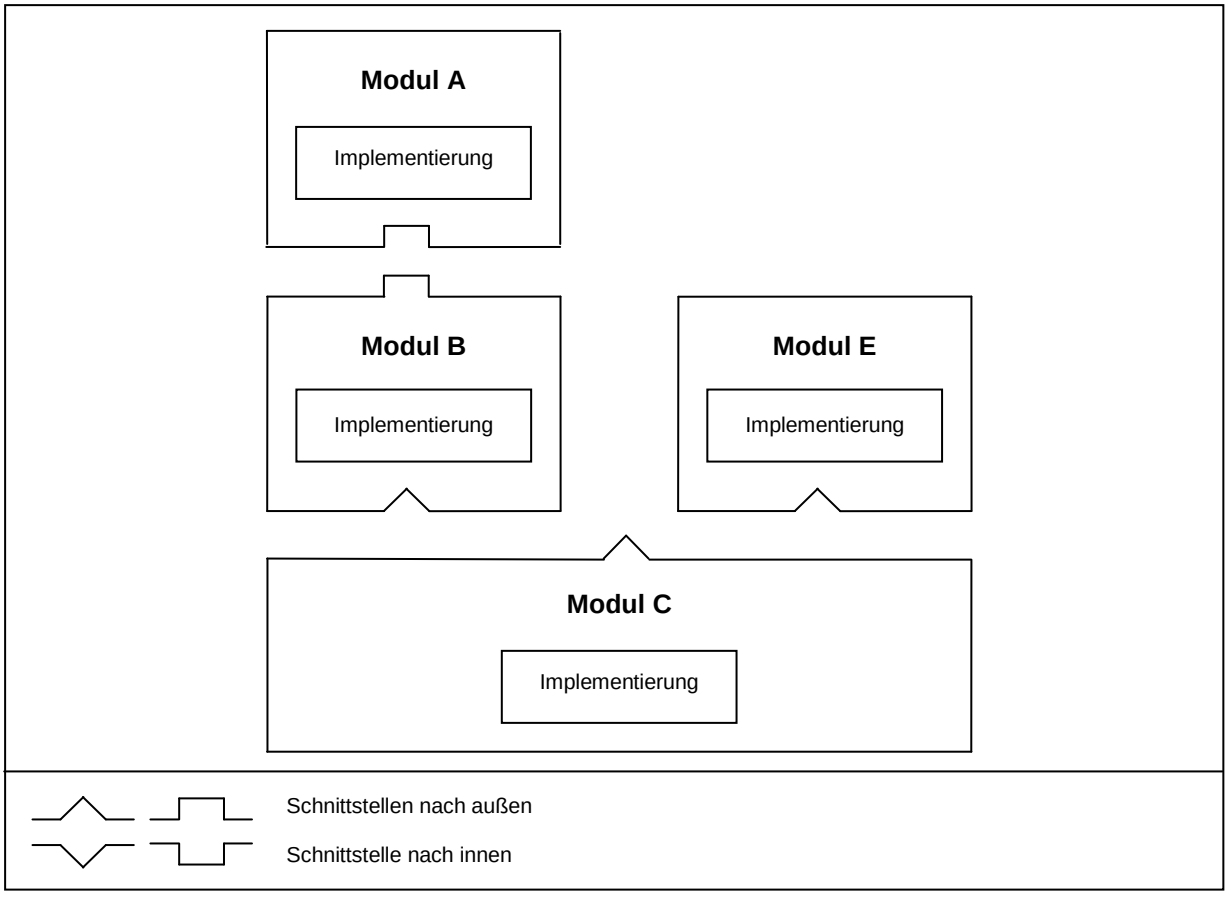



**Abbildung 2: Pluggable Konzept**

Wie in Abbildung 2 zu sehen ist, nutzt das Modul E die gleiche Schnittstelle nach innen wie das Modul C nach außen. Es nutzt die von C zur Verfügung gestellten Ressourcen. Somit kommuniziert Modul C jetzt indirekt mit beiden Modulen, B und E. Dadurch hat Modul E das System erweitert, ohne in die bisherigen Beziehungen einzugreifen.

Durch die bisher erwähnten Kriterien, wie zum Beispiel Austauschbarkeit, Erweiterbarkeit und die damit verbunden Abhängigkeiten zwischen den Modulen ist eine hierarchische Zerlegung und eine Trennung in verschiedene Schichten notwendig. Dies soll dabei helfen, bei wachsender Modulanzahl und damit auch wachsender Anzahl an Beziehungen und Abhängigkeiten, klare Strukturen zu schaffen.

## 1.2 Shop-System

Softwaregrundlage für jeden Online-Shop, auch als E-Shop bezeichnet, ist ein Shop-System. Derzeit gibt es in Deutschland über 200 Shop-Softwareanbieter. Diese unterscheiden sich in der Ausrichtung ihres Angebotes, sowie in der Orientierung auf verschiedene Zielgruppen. Es gibt Shop-Systeme für den Massenmarkt, aber auch Anbieter die sich spezialisieren und auf den Kunden zugeschnittene individuelle Softwarelösungen entwickeln. Es gibt Shop-Systeme, die für den deutschen Markt entwickelt werden, andere die auf den französischen Markt angepasst sind. Es gibt kostenlose Open-Source Lösungen, teurere Premium-Lösungen und Enterprise-Lösungen. Aber der Hauptteil dieser Shop-Systeme hat einen gemeinsamen Nenner – ihren Aufbau.

### 1.2.1 Aufbau

Prinzipiell kann gesagt werden, dass ein Shop-System aus den folgenden Komponenten besteht.[1]

- Datenhaltung
- Administrationsbereich
- Präsentationssystem
- Payment-Gateway
- Werkzeuge

Für die Datenhaltung werden in den meisten Fällen Datenbanken benutzt, da diese nicht nur schnell sind, sondern auch eine Sicherung der Daten bieten. Zudem stellen viele Datenbanken eine Nutzerverwaltung zur Verfügung.
Im Administrationsbereich ist es dem Shop-Betreiber möglich, verschiedenste Einstellungen am System vorzunehmen. Der Umfang dieser Einstellungen variiert dabei sehr stark. Zusätzlich bieten die meisten Shop-Systeme eine passende Schnittstelle zur Datenbank. Dadurch ist es zum Beispiel möglich, neue Produkte in das System einzupflegen, oder schon vorhandene Produktdaten zu aktualisieren. Da die vorhanden Funktionen von Shop-System zu Shop-System sehr stark variieren, wird auf weitere Beispiele verzichtet. Festzuhalten ist aber, dass der Administrationsbereich dazu, dient den Onlineshop zu verwalten und zu konfigurieren. Desweiteren ist noch anzumerken, dass sehr viele Shop-Systeme auf eine grafisch gestützte Verwaltung Wert legen, um eine einfache Bedienung für den Shop-Betreiber zu gewährleisten. Das heißt, sie treten meist als eigenständige Webanwendungen

[1] In Anlehnung an: KUPPER, RAMON (2011)

auf. Sie sind lokal und/oder via Internetverbindung über einen Internetbrowser aufrufbar und bedienbar.
Das Präsentationssystem stellt das eigentliche Gesicht des Shop-Systems dar. Es ist sozusagen die Schnittstelle zum Kunden, das heißt der einzige Bereich des gesamten Systems, mit dem der Kunde in Kontakt kommt. Es handelt sich hier um eine oder mehrere zusammengehörige dynamische Webanwendungen. Oder es handelt sich um zusammengehörige statische Internetseiten. Oft sind auch diese beiden Sachen miteinander vermischt. Um es auf das Wesentliche zu reduzieren, kann man sagen, dass die Aufgabe der Präsentationsschicht darin besteht, einen elektronischen Shop mit Produkten darzustellen und den potentiellen Kunden, anhand der grafischen Darstellung zu überzeugen, ein Produkt zu kaufen. Die Art und Weise der grafischen Darstellung der Daten und der zusätzlichen Funktionen ist auch wieder sehr stark vom Shop-System abhängig. Die eigentlichen Abläufe und Geschäftsmethoden liegen nicht im Präsentationssystem, es spiegelt lediglich die Geschäftslogiken des Shop-Systems wider.
Aufgabe des Payment Gateways ist es zum Beispiel, bei einer Bestellung des Kunden mit einer Kreditkarte diese elektronische Transaktion durchzuführen und auf ihre Gültigkeit hin zu überprüfen.
Werkzeuge eines Shop-Systems können sein, zum Beispiel der Import von Produktkatalogen und bei Aktionen, die Generierung von Gutscheinen und eine automatische Email-Benachrichtigung der registrierten Kunden.

### 1.2.2 Einteilung

Vereinfacht gesehen kann man ein Shop-System auf zwei grundlegende Komponenten eingrenzen. Einen Part der näher am Benutzer ist, dem Front-End. Und einem Part der näher am System liegt, dem Back-End.

#### Front-End

Das Front-End stellt die Komponente Präsentationssystem dar. So gesehen ist es also die Oberfläche zum Einkaufen. Neben der Präsentation der Produkte beinhaltet es noch weitere Funktionen. Wie zum Beispiel das Darstellen eines virtuellen Warenkorbes. Dieser stellt die ausgewählten Produkte des Benutzers, beziehungsweise des Kunden dar. Des Weiteren beinhaltet das Frontend den Prozess des Bestellvorgangs. Dies bedeutet, dass es die Rahmenbedingungen für die Bestellung, die Kundendaten und Bestelldaten sowie die Bestellung selbst an das Backend übermittelt. Klassischerweise ist der Bestellprozess in mehrere Bereiche aufgeteilt. Zum Einen das Erfassen der Bestelldaten, wie zum Beispiel Rechnungsadresse und Lieferadresse. Zum Anderen das Auswählen der Lieferbedingungen

und Versandbedingungen und letztendlich der Auswahl einer Zahlmethode, wie zum Beispiel Nachnahme oder Vorkasse, und die Auswahl des Versanddienstes. Sind die Rahmenbedingungen ausgewählt und die Daten erfasst, erfolgt nun die Bestellübersicht. Der nächste Schritt ist das automatische Versenden der Bestellbestätigung in Form einer E-Mail. Verbunden damit ist meist noch eine grafische Anzeige, dass die Bestellung erfolgreich war.
Außer Acht gelassen wurde zum Beispiel die rechtliche Zustimmung des Kunden, dass er die Widerrufsbelehrung gelesen hat. Da es sich beim Frontend so gesehen um eine Website handelt, müssen noch weitere rechtliche Gegebenheiten erfüllt werden. Dies betrifft nicht nur den Bestellprozess. Es wird darauf aber nicht weiter eingegangen, es soll lediglich der Vollständigkeit halber erwähnt werden. Während des Bestellprozesses hat das Front-End mehrmals mit dem Back-End kommuniziert. Zum Beispiel ging die Berechnung des tatsächlichen Endpreises vom Back-End aus. Auch für das automatische Versenden der E-Mail war das Backe-End zuständig. Dies bedeutet das, dass Front-End fast keine, oder gar keine Geschäftslogiken enthält.

**Back-End**

Wie schon im Punkt 1.2.1 Aufbau erwähnt, lässt sich der Shop mit Hilfe des Back-Ends verwalten. Es lassen sich damit Kategorien für Produkte anlegen, löschen und ändern, Bestellungen bearbeiten, Zahlungsarten und Versandarten festlegen. Die Funktionen sind abhängig vom jeweiligen Shop-System und lassen sich damit nur schwer zusammenfassen. Dies kann mit dem Administrationsbereich, im oberen Punkt 1.2.1 Aufbau, vergleichen werden.

## 1.3 OSGi

Die OSGi Alliance wurde im März 1999 gegründet. Sie ist ein internationaler, nicht-kommerzieller Verbund dessen Mitglieder nicht nur aus Großunternehmen bestehen, wie zum Beispiel der Deutschen Telekom AG, Oracle Corporation und SAP AG. Auch staatliche Organisationen, Bildungseinrichtungen und jegliche sonstige Unternehmen können Mitglied werden, sofern sie mit den Zielen, Prozessen und Grundsätzen der Allianz übereinstimmen.
Der Begriff OSGi selbst steht für *Open Services Gateway Initiative*. Er ist mittlerweile das Synonym geworden für die Idee, die hinter der OSGi Alliance steht. Nämlich eine offene Spezifikation zu entwerfen, die es erlaubt, modulare Software, mit Hilfe von Java Technologien, zu entwickeln. Die OSGi Alliance entwirft lediglich die Spezifikation. Außerdem stellt sie eine Referenzimplementierung bereit, diese soll lediglich als Beispiel und Orientierung dienen. Bekannte und weitverbreitete OSGi Frameworks, die diese Spezifikationen implementieren, sind zum Beispiel Equinox[2], Spring DM[3], Apache Felix[4] und Knopflerfish[5]. Desweiteren stellt die Allianz Test-Suiten und Zertifizierungen bereit. Damit soll eine hardwareunabhängige und dynamische Softwareplattform geschaffen werden, welche durch das Komponentenmodell einen modularen Aufbau bietet.[6]

### 1.3.1 Architektur

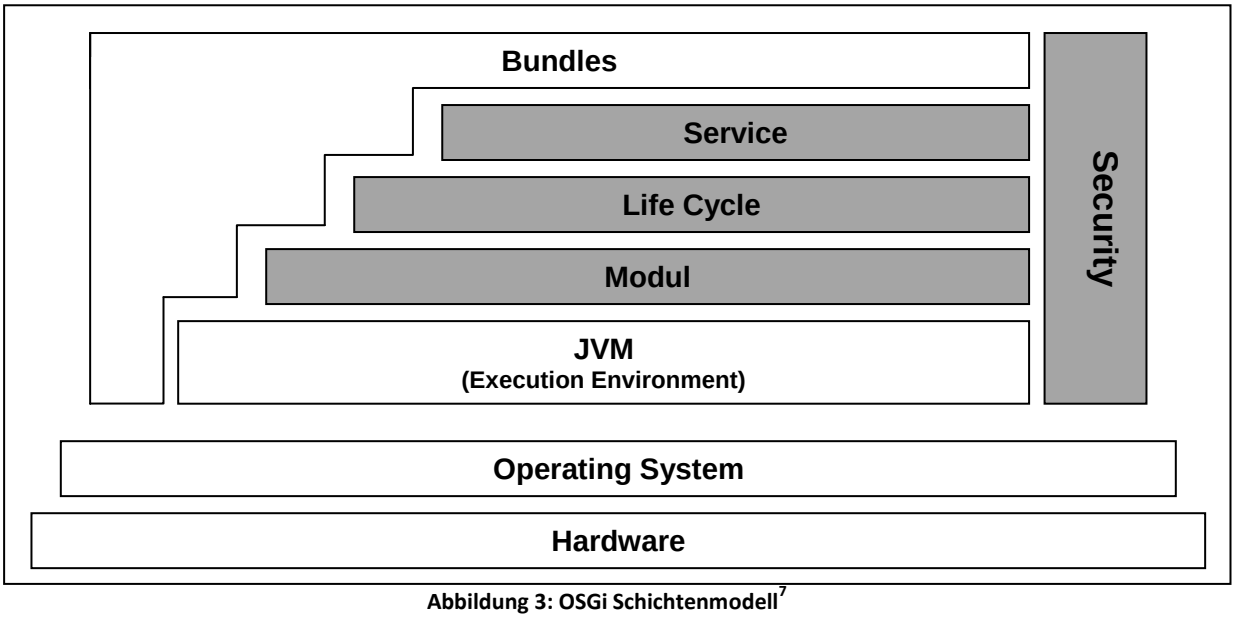



**Abbildung 3: OSGi Schichtenmodell[7]**

[2] http://www.eclipse.org/equinox/
[3] http://www.springsource.org/osgi
[4] http://felix.apache.org/site/index.html
[5] http://www.knopflerfish.org/
[6] Quelle: OSGi Alliance (2011a); OSGi Alliance (2011b); OSGi Alliance (2011c)
[7] In Anlehnung an: FUNKE, HOLGER (2009), Abb. 1; SEEBERGER, HEIKO (2008), Abb. 1

Nachfolgend wird das Schichtenmodell der OSGi-Architektur anhand der voranstehenden Abbildung 3 näher erläutert. Die grau markierten Rechtecke sind dabei von großer Relevanz. Denn diese stellen die Schichten des OSGi-Frameworks dar. Ganz unten befindet sich die Hardware-Schicht. Diese ist grundlegend erforderlich, weil darauf die weiteren Schichten aufbauen. Darüber befindet sich die Betriebssystem-Schicht und darauf laufend die JVM, da OSGi nur in einer Java-Laufzeitumgebung funktioniert.
Jedes Bundle besitzt eine sogenannte Manifest-Datei. Mit dieser Datei lassen sich explizit die Bundle-Abhängigkeiten festlegen und unter Anderem lässt sich damit auch genau eine Java-Laufzeitumgebung festlegen. Hier kommt schon die Modul-Schicht zum Tragen.

**Modul-Schicht**

Diese Schicht ist für den strukturellen Aspekt des OSGi-Modulkonzeptes zuständig. Sie definiert ein OSGi-Bundle, auch als Modul bezeichnet, als kleinste logische Einheit.
Wie schon oben erwähnt, wird in dieser Schicht unter Anderem beschrieben, welche Ressourcen, Klassen und Services ein Bundle exportiert und importiert. Abgebildet wird dies durch die Manifest-Datei. Sie kann auch weitere Einträge beinalten, wie zum Beispiel den Namen des Bundle, die erforderliche Java-Laufzeitumgebung, die Bundle-Version usw. Eine Liste der Manifest Einträge ist unter folgenden Link auf der offiziellen OSGi Alliance Website einzusehen[8]. Dabei ist aber zu beachten, dass nicht alle dieser aufgelisteten Einträge der OSGi-Spezifikation entsprechen.[9] Nachfolgend ein Beispiel einer Manifest-Datei.

```
01  Manifest-Version: 1.0
02  Bundle-ManifestVersion: 2
03  Bundle-Name: beispiel Bundle
04  Bundle-SymbolicName: beispiel.bundle
05  Bundle-Version: 1.0.0.qualifier
06  Bundle-Activator: beispiel.bundle.activator.Activator
07  Bundle-ActivationPolicy: lazy
08  Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

**Listing 1: Beispiel einer Manifest-Datei**

Der Verfasser dieser Bachelorarbeit weist darauf hin, dass die Einträge in der Manifest-Datei der OSGi Spezifikation entsprechen müssen. Je nach verwendetem OSGi-Framework können weitere Einträge hinzukommen. Ein Beispiel wäre Equinox, das weitere Manifest-Einträge zulässt.

[8] http://www.osgi.org/Specifications/ReferenceHeaders
[9] In Anlehnung an: SEEBERGER, HEIKO (2008)

### Lebenszyklus-Schicht

Oberhalb der relativ statischen Modul-Schicht befindet sich die dynamischere Lebenszyklus-Schicht.[10] Diese beinhaltet, wie der Name schon vorwegnimmt, den Lebenszyklus eines Modules, beziehungsweise eines Bundle. Folgende Abbildung stellt dies grafisch dar.

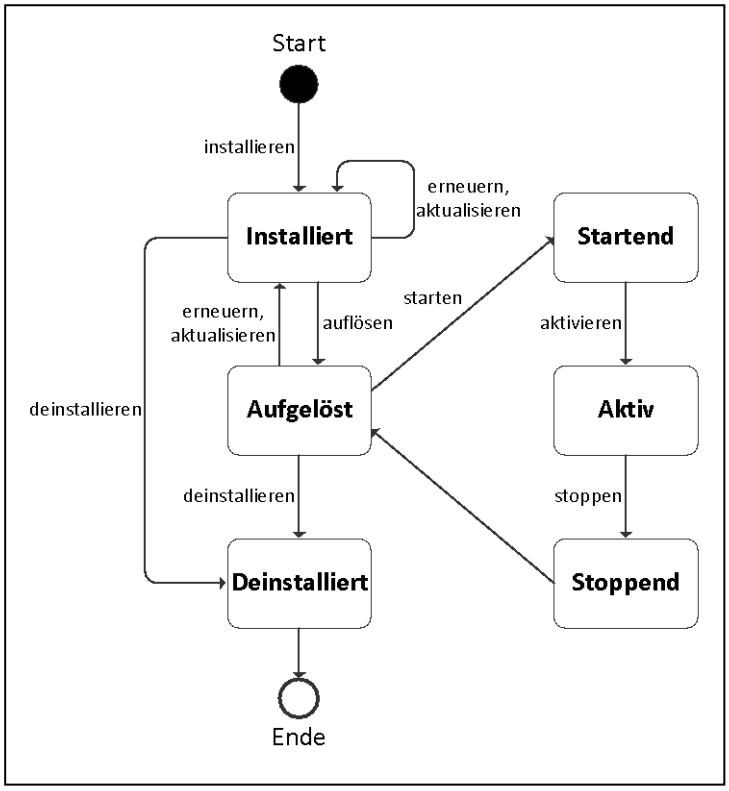



**Abbildung 4: Lebenszyklus eines OSGi Bundle**[11]

Der erste Schritt ist immer das Installieren des Bundle im OSGi-Framework. Prinzipiell gibt es zwei Methoden dafür. Zum einen über die OSGi-Konsole. Dort wird der Pfad zur Datei angegeben, die auf dem lokalen System liegt. Dieser Pfad kann sowohl absolut als auch relativ sein. Es ist aber auch möglich, ein Bundle über die Eingabe einer URL zu installieren. Beide Varianten werden mit dem *install* Befehl ausgeführt.

Die OSGi-Konsole ist eine einfache Textkonsole, welche vergleichbar mit der bekannten Java-Konsole ist. Auf dieser Konsole können OSGi-spezifische Befehle eingegeben werden, wie zum Beispiel *help*, *start*, *stop*, *update*, *refresh*, *close* und *ss*, Abkürzung für short status. Zusätzlich werden Java-Meldungen angezeigt, wie zum Beispiel Fehlermeldungen. Die OSGi-Konsole ist bei den meisten OSGi-Implementierungen vorhanden und standardmäßig aktiviert.

[10] In Anlehnung an: SEEBERGER, HEIKO (2008)
[11] In Anlehnung an: FUNKE, HOLGER (2009), Abb. 2

Zusätzlich zur OSGi-Konsole, kann man ein Bundle auch programmatisch installieren. Dies bedeutet, dass ein aktives Bundle über dessen eigenen Code ein oder mehrere Bundle installieren kann. Ist das Bundle einmal installiert bedeutet dies nur, dass es alle erforderlichen Formalitäten mit sich bringt und bereit ist, in den nächsten Zustand überzugehen.

Ausgehend vom ersten Zustand *Installiert* kann ein Bundle gleich wieder deinstalliert werden. Dies kann analog wie beim Installieren eines Bundle programmatisch gelöst werden. Die andere Möglichkeit wäre wieder über die OSGi-Konsole. Der Befehl dafür lautet *uninstall*. Danach muss dann entweder die Bundle-ID folgen, welche eine oder mehrere Ziffern beinhaltet, oder muss der voll qualifizierte Name des Bundle angegeben werden. Da ein Bundle letztendlich eine Jar-Datei ist, muss der Name dieser Datei angegeben werden.

Ausgehend vom Status *Installiert* kann ein Bundle nicht nur in den Status *Deinstalliert* wechseln, sondern auch in den Status *Aufgelöst*. Das bedeutet, die Abhängigkeiten eines Bundles wurden erfolgreich aufgelöst. Gegen eine erfolgreiche Auflösung würde zum Beispiel sprechen, dass ein Bundle A eine Klasse von einem anderen Bundle importiert. Wenn aber das andere Bundle selbst noch nicht im minimal erforderlichen Zustand *Installiert* ist, oder das erforderliche Bundle die geforderte Klasse nicht exportiert, kann die Abhängigkeit nicht erfolgreich aufgelöst werden. Damit bleibt A im Zustand *Installiert*.

Konnte ein Bundle erfolgreich aufgelöst werden, kann es in zwei weitere Zustände wechseln. Zum einen in den Zustand *Deinstalliert* und zum anderen in den Zustand *Startend*. Bei letzterem werden alle benötigten Klassen des Bundle geladen. Durch die Option *lazy* in der Manifest Datei kann der Zustand *Startend* künstlich verlängert werden. Das Bundle verbleibt solange in diesem Status, bis der Service, den das Bundle anbietet, von einem anderen Bundle genutzt wird.
Somit wird es auf den eigentlichen Betrieb vorbereitet. Von diesem Zustand aus gibt es kein direktes Zurück mehr in den Zustand *Aufgelöst*. Dies ist nur indirekt über die beiden Zustände *Aktiv* und *Stoppend* möglich.

Durch den Befehl *start* in der OSGi-Konsole geht ein Bundle vom Zustand *Aufgelöst* in Zustand *Aktiv*. Dabei passiert es zwangsweise den Zustand *Startend*. Ist ein Bundle aktiv bedeutet dies, dass es erfolgreich installiert wurde, alle Abhängigkeiten aufgelöst werden konnten und es im eigentlichen Betrieb ist. Konnte das Bundle nun aber nicht in den Zustand *Aktiv* gehen, weil im vorherigen Zustand das Laden einer Klasse fehlschlug, geht es über den Zustand *Stoppend* direkt wieder in den Zustand *Aufgelöst* über.

Damit ein Bundle von Aktiv in einen anderen Status wechseln kann, muss es über den Zustand *Stoppend* gehen. Und von da aus zu *Aufgelöst*. Der Befehl um ein aktives Bundle in diesen Status zu überführen lautet *stop* und nachfolgend die Bundle-ID oder der voll qualifizierte Name des Bundles.

In der oberen Abbildung 4 Lebenszyklus eines OSGi Bundle ist das Aktualisieren eines Bundle nicht ganz ersichtlich für den Betrachter. Aktualisiert man ein Modul, welches gerade aktiv ist, wird das betreffende Bundle gestoppt und wieder gestartet. Der eigentliche Sinn des Aktualisierens ist es aber das bestehende Bundle mit einer neueren Version zu ersetzen. Die ist in den Zuständen *Installiert*, *Aufgelöst* und *Aktiv* möglich.
Erneuert man ein aktives Bundle wird es gestoppt. Die Abhängigkeiten werden neu aufgelöst, das heißt im Falle einer aktualisiert bereitgestellten Jar-Datei wird diese neu in den Speicher der JVM geladen und damit die vorherige Version ersetzt. Anschließend wird das Bundle wieder gestartet.

**Service-Schicht**

Die Service Schicht bietet eine Reihe von Mechanismen, welche dazu dienen, ein dynamisches Verbundmodell zu schaffen, das zu dem noch sehr eng mit der Lebenszyklusschicht verbunden ist. Dadurch wird ein Service-Modell umgesetzt, welches auf folgenden Prinzip beruht: Ein Service wird veröffentlicht, er wird gefunden und gebunden.

Stellt man sich einmal vor, dass ein Bundle einen Service oder mehrere Services anbietet. Wie schon in den vorherigen Kapiteln erläutert wurde, kann ein Bundle zur Laufzeit installiert werden und auch wieder deinstalliert werden. Auch Dienste, die ein Bundle anbietet, können zu jeder Zeit registriert und wieder abgemeldet werden. Auch ein Bundle, was diese angebotenen Services nutzt, muss jeder Zeit damit rechnen, dass sich der Dienst ändert, oder auf einmal nicht mehr verfügbar ist.[12]

Dafür gibt es wie schon erwähnt eine Reihe von Mechanismen, um mit dieser Thematik umzugehen. Zu einem späteren Zeitpunkt wird darauf genauer eingegangen. Zum Beispiel wie ein Service statisch und dynamisch angemeldet und wieder abgemeldet werden kann und wie ein Service im OSGi-Kontext definiert ist.

[12] In Anlehnung an: FUNKE, HOLGER (2009)

**Sicherheits-Schicht**

Vertikal zu den bisher vorgestellten Schichten Modul, Lebenszyklus und Service befindet sich die Sicherheits-Schicht. Dies baut auf dem in Java2 eingeführten Sicherheitsmodel auf. Dieses Modell wird erweitert, um den OSGi-spezifischen Anforderungen Rechnung zu tragen. Zum Beispiel beinhaltet diese Schicht Mechanismen, die definieren, welche Ausführungsrechte ein Bundle besitzt, welchen Service ein Bundle benutzen darf und auf welche Ressourcen eines anderen Bundle es zugreifen kann.[13]

### 1.3.2 Aufbau eines OSGi-Bundle

Es stellt sich die Frage, was ein Bundle eigentlich ist, beziehungsweise wie es aufgebaut ist. Im Punkt 1.3.1 Architektur wurde es wie folgt definiert: Ein Bundle ist die kleinste, logische Einheit im Gesamtsystem.
Zudem wurde im vorherigen Punkt 1.3.1 festgestellt, dass ein Bundle ein oder mehrere Services anbieten kann und Services von einem, oder mehreren Bundle nutzen kann.
Laut OSGi Spezifikation, ist ein Bundle eine normale JAR-Datei, welche im standardmäßigen ZIP-basierenden Dateiformat vorliegt. Der Name dieser Datei setzt sich im Normalfall aus dem Bundle-Namen zusammen, gefolgt von einer OSGi-gemäßen Versionsangabe und der Dateiendung *.jar*. JAR-Dateien können wiederum andere JAR-Dateien enthalten. Zum Beispiel Java-Bibliotheken von Drittherstellern.
Außerdem liegt der eigentliche kompilierte Java-Quellcode des Bundle in seiner JAR-Datei.
Des Weiteren kann ein Bundle Ressourcen enthalten, wie zum Beispiel Bilder, Webseiten Dateien, Dokumente usw. Generell muss ein Bundle eine Manifest Datei beinhalten. Diese ist für die Beschreibung des Bundle zuständig. Zum Beispiel, wie das Bundle heißt, welche Version des Bundle vorliegt, welche externen Bibliotheken es benötigt, welche Ressourcen es von sich freigibt usw. Klassischerweise gibt es im Bundle einen Ordner namens *META-INF* und in diesem Ordner ist in den häufigsten Fällen nur eine Datei vorhanden, mit dem Namen *MANIFEST.MF*, welche die Manifest-Datei des Bundle ist. Über diese spezielle Datei wurde schon in einem der vorherigen Punkte gesprochen.
Der Vollständigkeit halber sei erwähnt, dass eine Bundle-JAR-Datei zu dem noch einen Ordner mit dem Namen OSGI-OPT besitzen kann. In diesem Ordner können jegliche Arten von Informationen in Form einer Datei hinterlegt werden. Zum Beispiel könnte es dazu benutzt werden, um den blanken Quelltext des Bundle zu speichern. [14]

[13] In Anlehnung an: SEEBERGER, HEIKO (2008)
[14] Vgl. OSGi Alliance (2009), S. 27 ff.

### 1.3.3 OSGi Registry

Im Kapitel 1.3.1 Architektur wurde erläutert, dass ein Bundle, Services registrieren, wieder abmelden und selber konsumieren kann. Dafür gibt es im OSGi-Framework die Service-Registry. Sie ist sozusagen die zentrale Anlaufstelle für alle Services. Denn ein Bundle holt sich den Service eines anderen Bundle nicht direkt von diesem, sondern, es holt sich den Service über die Registry. Dasselbe gilt für das Registrieren eines Services. Dieser wird global im OSGi-Framework in der Registry angemeldet. Damit hat prinzipiell jedes Bundle Zugriff auf einen Service über die Registrierung.
Dabei sei angemerkt, dass dies nicht automatisch bedeutet, dass das Bundle etwas mit dem Service anfangen kann, beziehungsweise es den Service auch wirklich nutzen kann. Dies wäre zum Beispiel der Fall, wenn ein Bundle einen Service benutzen möchte, es aber das Java-Objekt nicht kennt, durch was der Service definiert ist.

### 1.3.4 BundleActivator und BundleContext

Ein wichtiger Bestandteil, eines fast jeden Bundle, ist der Bundle-Activator. Dieser ist eine Java-Klasse mit beliebigen Namen, welches das BundleActivator-Interface implementiert. Das Interface beinhaltet nur zwei Methoden. Eine für das Starten des Bundle und eine Methode für das Stoppen des Bundle.
Ein Bundle kann zwar mehrere Activator-Klassen besitzen, aber nur eine davon benutzen. Der Grund liegt darin, dass die Activator Klasse, mit dem Eintrag *Bundle-Activator* in der Manifest-Datei des Bundle, angegeben werden muss. Hier muss der voll qualifizierte Name angegeben werden. Zum Beispiel *osgi.beispiel.BeispielActivator*. Es ist nicht möglich, mehrere Bundle-Activator-Klassen anzugeben.
Ein Bundle wechselt erst dann von dem Zustand *Startend* in den Zustand *Aktiv*, wenn zuvor das Bundle-Activator-Objekt erfolgreich vom OSGi-Framework instanziiert werden konnte. Vorausgesetzt, das Bundle hat eine Activator-Klasse und diese ist in der Manifest-Datei im Eintrag *Bundle-Activator* definiert. Beim Wechseln zwischen den beiden Zuständen wird die start()-Methode aufgerufen, die das Bundle-Activator-Interface definiert. Wirft die Start-Methode eine Exception, wechselt das Bundle in den Zustand *Stoppend* und danach gleich in den Zustand *Aufgelöst*. Dabei ruft das OSGi-Framework nicht zwingend die Stop-Methode auf. Diese vom BundleActivor-Interface definierte stop()-Methode wird dann aufgerufen, wenn das Bundle erfolgreich im Zustand *Aktiv* ist und gestoppt wird. Sinn dieser Methode ist es, alle von der start()-Methode getätigten Aktionen rückgängig zu machen. Dabei ist es unnötig, sich um das Abmelden von Services und Framework-Listener zu kümmern, da das Aufräumen vom Framework selbst übernommen wird.

Nachfolgend ein Programmcodebeispiel einer BundleActivator-Klasse.

```
01    package osgi.beispiel;
02    import org.osgi.framework.BundleActivator;
03    import org.osgi.framework.BundleContext;
04    public class BeispielActivator implements BundleActivator {
05            public void start(BundleContext bundleContext) throws Exception {
06                    // Programmcode
07            }
08
09            public void stop(BundleContext bundleContext) throws Exception {
10                    // Programmcode
11            }
12    }
```

**Listing 2: BundleActivator Beispiel**

Die start()- und stop()-Methode erwartet jeweils einen Paramter *bundleContext* vom Typ OSGi-*BundleContext*. Damit bekommt jedes Bundle seinen eigenen, vom Framework erzeugten, OSGi-BundleContext. Über dieses erzeugte Objekt kann man zum Beispiel den Namen des Bundle herausfinden, die Version, den Zustand usw. Zudem lassen sich über den BundleContext Services registrieren und abmelden. Damit fungiert der BundleContext als Schnittstelle zwischen dem Bundle und dem OSGi-Framework.

### 1.3.5 Services

In diesem Abschnitt wird erläutert, was ein OSGi-Service ist, wie und wo man ihn anmeldet und wieder abmeldet und welche Mechanismen für den Zugriff auf einen OSGi-Service durch ein Bundle existieren.

Die OSGi-Spezifikation definiert einen Service folgendermaßen: Ein Dienst ist ein normales Java-Objekt, welches unter einem, oder mehreren Java-Interfaces in der Service Registrierung angemeldet wird.[15]
In der folgenden Darstellung wird das allgemeine Registrieren eines Services über ein Java-Interface abgebildet.

[15] Vgl. OSGi Alliance (2009), S. 123

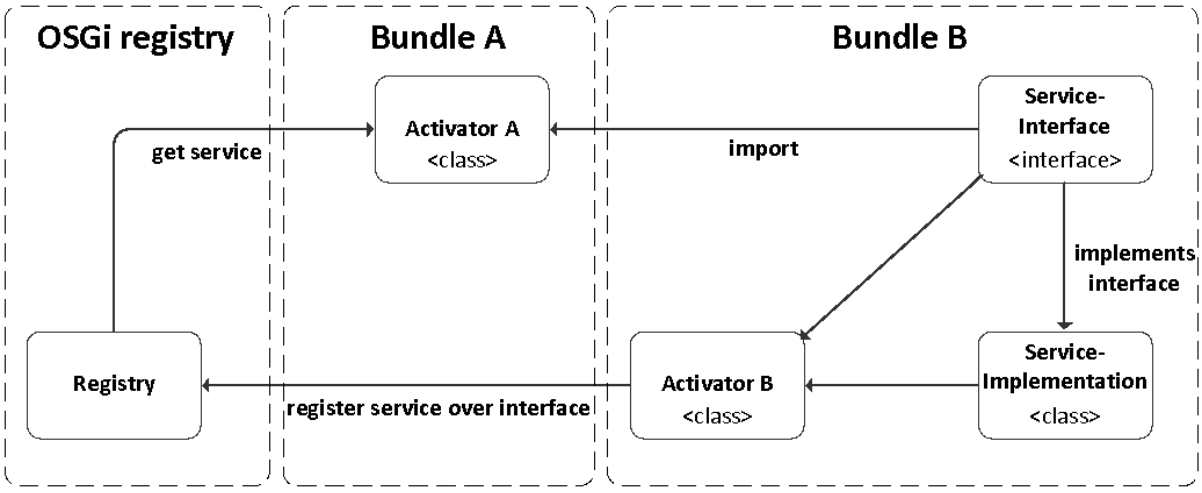



**Abbildung 5: Registrierung eines OSGi Service unter einem Java-Interface**

In den vorherigen Abschnitten wurde erläutert, was die OSGi-Registry ist und was ein Bundle-Activator ist. Wie zu erkennen, ist die Abbildung 5 dreigeteilt. Bundle A fungiert als Konsument, Bundle B als Produzent und die OSGi-Registry als Zwischenhändler. Bundle A konsumiert also einen Service den Bundle B anbietet.
B besitzt ein Service-Interface, welches vom Typ Java-Interface ist. Die Java-Klasse *Service-Implementation* ist eine Java-Klasse, die das *Service-Interface* implementiert. Im Activator B wird ein Service unter dem Namen des Service-Interfaces in der OSGI-Registry angemeldet. Und zusätzlich wird eine Implementation des Interfaces mit registriert. In dem Fall, ist dies die Klasse *Service-Implementation*. Zusätzlich exportiert Bundle B die Klasse Service-Interface und Bundle A importiert das Service-Interface von B. A fragt nun die OSGi-Registry, ob diese einen Service hat, der mit dem Namen des Service-Interfaces registriert ist. Damit bekommt A eine Referenz auf den Service zurück, falls der Service überhaupt verfügbar ist.
Der Verfasser der Bachelorarbeit merkt an, dass ein Service nicht zwingend unter einem Interface registriert werden muss. Auch normale Java-Klassen können als Service registriert werden, wie zum Beispiel java.lang.String, java.lang.Long und eigene Klassen. Allerdings geht damit die Trennung der Service-Definition und der konkreten Implementierung, sowie die Möglichkeit des einfachen Austauschens der Implementierung, verloren. Außerdem ist es auch nicht zwingend notwendig, den Service unter dem voll qualifizierten Namen des Service-Interfaces zu registrieren. Ein selbstdefinierter Java-String ist ebenso möglich, wie zum Beispiel *„MeinService“*. Dabei ist dann aber nicht mehr die eindeutige Zuordnung des OSGi-Services, anhand des Service-Namens, gegeben.

Nachfolgend werden verschiedene Mechanismen aufgezeigt, um auf einen Service Zugriff zu erhalten, der über ein Interface registriert ist.

Als Orientierung dient Abbildung 5: Registrierung eines OSGi Service unter einem Java-Interface. Bundle B besitzt folgende Komponenten:

**Service-Interface:**

Das Interface für den Service definiert eine Methode getName(), welche ein String-Objekt zurückliefert.

```
01  package osgi.beispiel.producer;
02  public interface ServiceInterface {
03          String getName();
04  }
```

**Listing 3: Beispiel eines Service-Interfaces**

**Service-Implementierung:**

Diese Klasse implementiert das Service-Interface und besitzt damit die Methode getName(). Als konkrete Implementierung wird „Beispiel Service“ als Rückgabewert definiert.

```
01  package osgi.beispiel.producer;
02  public class ServiceImpl implements ServiceInterface {
03          @Override
04          public String getName() {
05                  return "Beispiel Service";
06          }
07  }
```

**Listing 4: Beispiel einer Implementierung eines Service-Interfaces**

**Activator-Klasse:**

Der Activator implementiert das BundleActivator-Interface. Zuerst wird in der start()-Methode ein Interface-Objekt erzeugt mit der Service-Implementierung. Dann wird der Service über den Bundle-Kontext, mit dem Service-Interface-Namen und dem erzeugten Interface-Objekt, registriert.

```
01  package osgi.beispiel.producer;
02  import org.osgi.framework.BundleActivator;
03  import org.osgi.framework.BundleContext;
04  public class Activator implements BundleActivator {
05          public void start(BundleContext bundleContext) throws Exception {
06                  ServiceInterface service = new ServiceImpl();
07                  // registriere den Service unter dem Service-Interface-Namen
08                  bundleContext.registerService(ServiceInterface.class.getName(), service, null);
09          }
10  … }
```

**Listing 5: Statisches Registrieren eines Services**

Der Verfasser weist daraufhin, dass man einen Service mit weiteren Informationen registrieren kann. In Zeile 8 des Listing 5 kann anstatt *null* ein java.util.Dictionary-Objekt verwendet werden. Auch Klassen, die von Dictionary ableiten, sind möglich, wie zum Beispiel *java.util.Hashtable* und *java.util.Properties*.
Zudem existiert eine überlagerte Methode BundleContext.registerService(). Mit dieser Methode kann man mehrere Bezeichnungen für einen Service angeben.

Bundle A enthält eine Activator-Klasse, die das Bundle-Activator-Interface implementiert. Diese wird unter Anderem benötigt, um Zugriff auf registrierte Services zu erhalten. Nachfolgend werden verschiedene Möglichkeiten aufgezeigt, wie man auf einen Service zugreifen kann, der über ein Interface und dessen Namen registriert ist.

**Standard-Service**

Zuerst wird über den Bundle-Kontext eine Referenz auf den Service geholt, der mit dem Namen des Service-Interfaces registriert ist. Über den Bundle-Kontext und mit Hilfe der zuvor geholten Service-Referenz, wird der eigentliche Service geholt, beziehungsweise das mit dem Service registrierte Service-Objekt. Danach wird auf die Methode getName(), des Service-Objektes, zugegriffen. Das zurückgelieferte String Objekt wird dann auf der Konsole ausgegeben. Der statische Zugriff auf einen OSGi-Service wird durch das nachfolgende Listing dargestellt.

```
01  package osgi.beispiel.consumer;
02  import org.osgi.framework.BundleActivator;
03  import org.osgi.framework.BundleContext;
04  import org.osgi.framework.ServiceReference;
05  import osgi.beispiel.producer.ServiceInterface;
06  public class Activator implements BundleActivator {
07          @Override
08          public void start(BundleContext bundleContext) throws Exception {
09                  // hole Referenz auf den Service mit dem Interface-Namen
10                  ServiceReference serviceReference = bundleContext
11                                  .getServiceReference(ServiceInterface.class.getName());
12                  // hole Service über die Service-Referenz
13                  ServiceInterface service = (ServiceInterface) bundleContext
14                                  .getService(serviceReference);
15                  // hole String-Objekt vom Service-Interface-Objekt
16                  System.out.println(service.getName());
17          }
18  … }
```

**Listing 6: Statischer Service-Zugriff**

### Service-Tracker

Der OSGi-ServiceTracker ist eine Art Hilfsmittel für den Programmierer, mit diesem Tracker ist es möglich OSGi-Services zu *tracken*, also aufzuspüren. Zusätzlich enthält der ServiceTracker Mechanismen die darüber informieren, wann ein OSGi-Service verfügbar ist und ob sich ein OSGi-Service geändert hat.

Zuerst wird in Zeile 10, des noch folgenden Listing 7, ein neues ServiceTracker-Objekt erzeugt. In Zeile 13 wird der ServiceTracker dann mit der Methode open() geöffnet. Danach wird in der nächsten Zeile über das erzeugte ServiceTracker-Objekt der Service geholt. Anschließend wird auf die Mehtode getName(), des Service-Objektes, zugegriffen. Das zurückgelieferte String Objekt wird dann auf der Konsole ausgegeben.

```
01  package osgi.beispiel.consumer;
02  import org.osgi.framework.BundleActivator;
03  import org.osgi.framework.BundleContext;
04  import org.osgi.util.tracker.ServiceTracker;
05  import osgi.beispiel.producer.ServiceInterface;
06  public class Activator implements BundleActivator {
07          @Override
08          public void start(BundleContext bundleContext) throws Exception {
09                  // bilde neues ServiceTracker-Objekt
10                  ServiceTracker st = new ServiceTracker(bundleContext,
11                                  ServiceInterface.class.getName(), null);
12                  // öffne ServiceTracker
13                  st.open();
14                  // hole Service
15                  ServiceInterface service = (ServiceInterface) st.getService();
16                  // hole String-Objekt vom Service-Interface-Objekt
17                  System.out.println(service.getName());
18          }
19  … }
```

**Listing 7: Zugriff auf einen Service über ServiceTracker**

### Deklarative Services

Der Vollständigkeit halber soll auch die Verwendung von deklarativen Services kurz erwähnt werden. Ein deklarativer Service wird über eine XML-Datei beschrieben. Diese Datei muss im Bundle vorhanden sein, typischerweise im Ordner *OSGI-INF*. Außerdem muss die XML-Datei in der Manifest-Datei des Bundles, unter dem Eintrag *Service-Component* angegeben werden. Großer Vorteil dieser Methode ist, dass sowohl beim Registrieren des Services, als auch beim Verwenden des Services, eine dynamische Nutzung von OSGi-Services möglich ist.

# 2 Shop-System-Überblick

Das folgende Kapitel ist in mehrere Unterpunkte unterteilt. Im ersten Unterpunkt wird kurz beschrieben auf welche Weise eine Auswahl von Shop-Systemen erfolgt und wie dabei vorgegangen wurde. Im nächsten Schritt wird ein Vergleich dieser Systeme vorgenommen. Mittelpunkt der Betrachtung, ist der Bezug der Shop-Systeme zu einem modularen Softwaresystem. Nachfolgend werden die erhaltenen Ergebnisse ausgewertet und näher erläutert. Auf Grundlage der gewonnen Erkenntnisse werden Schlussfolgerungen gezogen. Aufbauend auf diesen Schlussfolgerungen, wird eine geeignete Technologie, für den zu entwickelnden Prototyp eines Shop-Systems, ermittelt.

## 2.1 Auswahl

Zuerst wurde ein grober Überblick verschafft, über bisher vorhandene Shop-Systeme. Grundlage dafür waren verschiedene Onlinerankings und Onlineshop-Listen, in denen zahlreiche Shop-Systeme aufgeführt sind. Bei dieser ersten Informationssichtung stellte sich heraus, dass es eine sehr große Auswahl an Shop-Systemen gibt. Daraufhin wurden stichprobenartig fünf Shop-Systeme ausgewählt und näher betrachtet. Dabei wurde bemerkt, dass die näher untersuchten Systeme sehr starke Unterschiede aufwiesen hinsichtlich ihrer Eigenschaften. Indikatoren dafür waren zum Beispiel der Preis, verwendete Technologien und beschriebene Funktionalitäten. Aus diesem Grund wurde entschieden, die Auswahl nach diesem ersten Überblick über verfügbare Shop-Systeme gezielt einzugrenzen. Dafür kommt ein Variantenvergleich zum Einsatz. Dadurch soll herausgefunden werden, welche Gemeinsamkeiten und Unterschiede es gibt, zwischen den Systemen und welche Variante, oder Varianten, den Ansprüchen eines modularen Softwaresystems am nächsten kommen, oder auch ganze erfüllt. Zudem soll dadurch ermittelt werden, welche Shop-System-Lösung, beziehungsweise welche Variante, für wen die passende ist. Also zum Beispiel, welche Variante, für ein mittelständisches Unternehmen, sinnvoll ist.

## 2.2 Variantenvergleich

Für den Vergleich von verschiedenen Varianten von Shop-Systemen wurde sich dafür entschieden, dies nicht anhand einer Pro- und Contra-Auflistung durchzuführen. Vielmehr werden Kriterien definiert, die direkt, oder indirekt darüber Aufschluss geben, welches Shop-System einem modularen Softwaresystem nahe kommt und welches Shop-System, für wen, eine gute Lösung darstellt. Zudem sollen die selbstdefinierten Kriterien Aufschluss über eventuelle Schwachstellen der Shop-Systeme geben.

Als erstes wurde der Preis als Kriterium ausgewählt. Dieser ist meist das entscheidende K.O.-Kriterium bei der Softwareauswahl, ob die Software überhaupt in Frage kommt oder nicht. Beim vorliegenden Vergleich spielt der Preis eher eine untergeordnete Rolle. Er soll lediglich als Indiz für die spätere Einordnung der Varianten in der Auswertung dienen.
Da in der näheren Betrachtung von Shop-Systemen, in der Vorauswahl, zu erkennen war, dass die Beschaffung von Informationen, über den genauen Aufbau der Shop-Systeme sehr schwierig sein könnte, beziehungsweise es im schlimmsten Fall überhaupt keine Informationen über den Aufbau gibt, wurde entschieden, Kriterien zu definieren, die lediglich indirekt darüber Aufschluss geben können, über den möglichen Aufbau. Beispiele dafür sind: Erweiterbarkeit, Skalierbarkeit, Mehrsprachigkeit. Weniger relevant für den Vergleich waren die eigentlichen Funktionen der Shop-Systeme, wie zum Beispiel ein komplexes Katalogsystem, Cross-selling, Nutzungsstatistiken, etc. Da diese Funktionen keinen wirklichen Aufschluss darüber geben, wie das Shop-System an sich arbeitet und wie es aufgebaut ist.

Im nächsten Schritt wurde die eigentliche Auswahl von Shop-Systemen getroffen, die für den Variantenvergleich in Frage kommen. Bei der Auswahl wurde darauf geachtet, ein möglichst großes und weites Spektrum abzudecken, um dadurch möglichst viele Varianten zu erhalten. Das heißt, angefangen bei den ganz kleinen Shop-System-Lösungen, bis zu den ganz großen Lösungen. Die erhobenen Daten der Untersuchungen wurden in einer Tabelle festgehalten. Diese befindet sich im Anhang auf Seite IX ff. Für eine bessere Übersichtlichkeit, wurde die Tabelle auf 3 kleinere Tabellen aufgeteilt.

## 2.3 Auswertung

Als nächstes wurden die erhobenen Daten analysiert und ausgewertet. Auf Basis der so erhalten Informationen, wurden die zuvor für den Vergleich gewählten Shop-Systeme untergliedert. Danach werden die durch die Auswertung des Vergleiches gewonnen Erkenntnisse näher erläutert. Auf dieser Grundlage aufbauend wird eine Technologieauswahl, für den zu entwickelnden Prototypen eines Shop-Systems, getroffen.

Für die Auswertung wurden die verschiedenen Shop-Systeme in vier Bereiche aufgeteilt. Diese Bereiche richten sich nach der Einteilung der Unternehmensgröße.

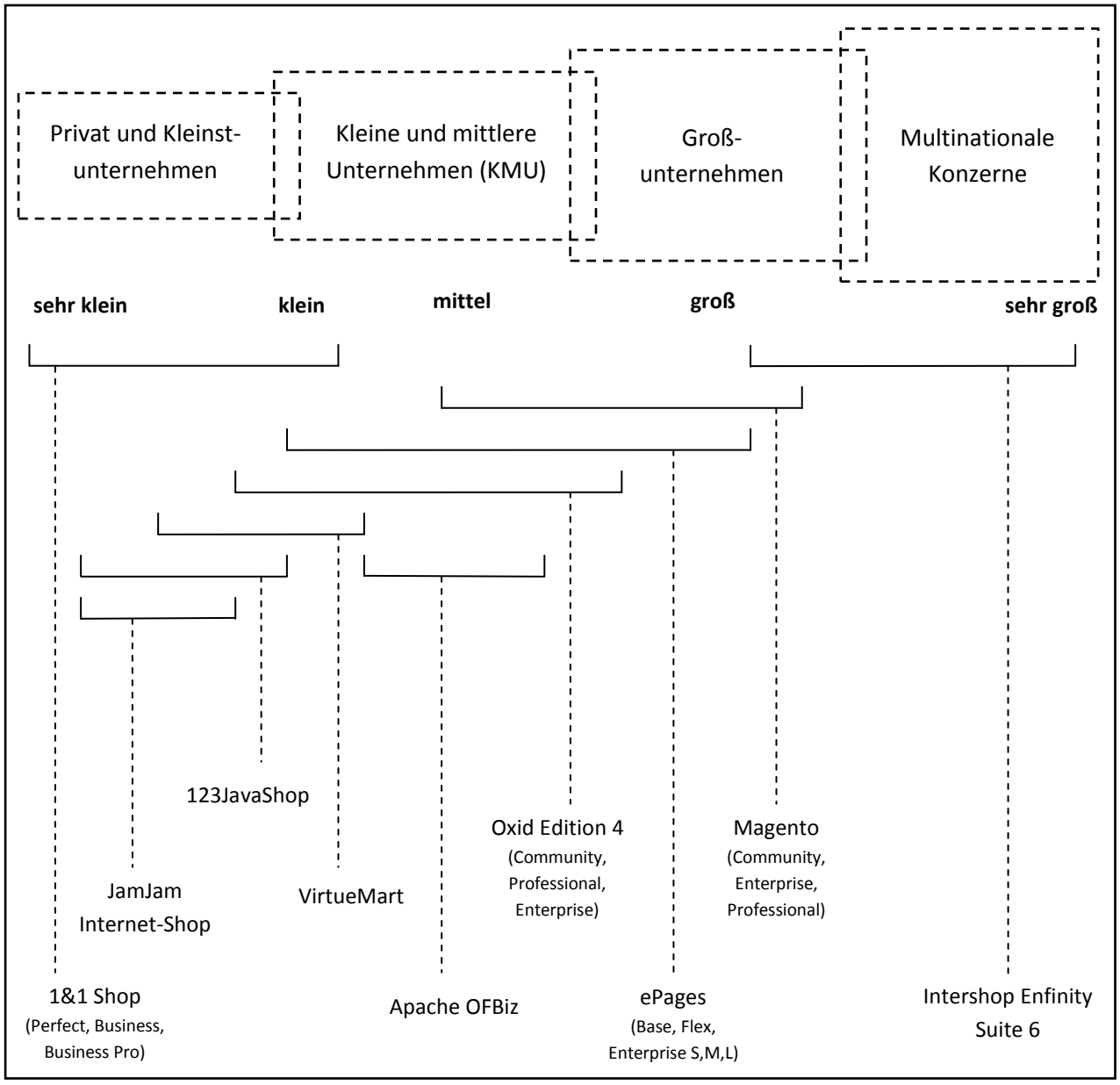



**Abbildung 6: Variantenvergleich - Einteilung Shop-Systeme**

Die Festlegung der Unternehmensgröße wurde anhand der folgenden Tabelle selbst definiert. Zusätzlich fand der private Sektor mit Beachtung.

| Unternehmen | Mitarbeiterzahl |
|---|---|
| Privat und Kleinstunternehmen | 0 - 50 |
| Kleine und mittlere Unternehmen (KMU) | 50 - 500 |
| Große Unternehmen | 500 - 5000 |
| Multinationale Konzerne | > 5000 |

**Tabelle 1: Einteilung Unternehmen**

Wie in Abbildung 6 zu erkennen ist, wurden die verschiedenen Shop-Systeme, die zuvor für den Vergleich ausgewählt wurden, in die Größe eines Unternehmens eingeordnet, beziehungsweise in die Spanne, in welcher die Shop-Systeme liegen.
Die Einordnung erfolgte anhand der verschiedenen Kriterien: Preis, vorausgesetztes (Programmier-, Technologie, Hintergrund) Wissen, Einrichtungsdauer, Benutzbarkeit, geschätzter Aufwand für Customizing, Shop-Funktionalitäten. Diese Informationen wurden ergänzend zum Variantenvergleich erarbeitet.
Zusätzlich wurden die verschiedenen Versionen der Shop-Systeme zusammengefasst, um einerseits, die Übersichtlichkeit zu wahren und andererseits, um eine deutlichere Darstellung zu erreichen, in welcher Spanne sich die Shop-Systeme bewegen. Voraussetzung hierfür ist, dass die einzelnen Shop-Systeme auch mehrere Versionen haben, die relativ nah beieinander liegen, um die Zusammenfassung der einzelnen Versionen zu rechtfertigen.
Zudem ist auf der Abbildung 6 zusehen, dass die Vierteilung der verschiedenen Unternehmensgrößen nicht klar abgegrenzt wurde. Der Grund liegt darin, dass die Grenzen zwischen den vier Einteilungen fließend sind und keine klare Abgrenzung zulassen. Somit ist die Einteilung in der oberen Tabelle nur als Orientierung zu sehen, um eine Vorstellung über die Größenverhältnisse zu erreichen.

Die eigentliche Auswertung beginnt mit den Gemeinsamkeiten zwischen den einzelnen Shop-Systemen. Es wurde festgestellt, dass fast alle Shop-Systeme eine unbegrenzte Anzahl an Produkten und Kunden zulassen. Wobei erst ab mittleren Unternehmensgrößen, beziehungsweise Shop-System-Größen große Produktzahlen und Kundenzahlen tatsächlich relevant sind. Ab diesen Größen stehen typischerweise technologische Mittel zur Verfügung, wie zum Beispiel Datenbankcluster, die mit großen Datenmengen umgehen können.
Eine weitere Gemeinsamkeit ist, dass im gesamten Spektrum die Möglichkeit der Mehrsprachigkeit vorhanden ist. Es zeichnet sich die Tendenz ab, dass bei den kleineren Shop-Systemen nur eine begrenzte Anzahl von Sprachen verfügbar ist und mit

zunehmender Größe diese Anzahl steigt. Es konnte aber nicht ermittelt werden, ob nur die reine Einstellung verschiedener Sprachen möglich ist, oder ob gleichzeitig länderspezifische Eigenheiten mit der Änderung verbunden sind. Zum Beispiel: Berücksichtigung der Währung, Formatierung (Preis, Textrichtung, Textzeichen) und verwendbare Zahlungsmethoden und Liefermethoden).

Zudem konnte festgestellt werden, dass im gesamten Spektrum die Shop-Systeme durch teilweise kostenpflichte Zusatzmodule, -erweiterungen, -funktionen ausbaufähig sind. Aber auch selbstentwickelte Erweiterungen sind möglich, zum Beispiel bei Magento[16], Apache OFBiz[17] und Oxid[18]. Der 1&1 Shop[19] bildet hier eine Ausnahme. Dieser ist aber erweiterbar auf eine höhere Stufe. Bei anderen Shop-Systemen mit Staffelungen wurde festgestellt, dass die Open-Source Lösung nicht erweiterbar ist, jedoch die höhere und damit kostenpflichtige Version. Beispiele sind hierfür Magento und Oxid.
Im Punkt Technologie wurde herausgefunden, dass viele der ausgewählten Systeme auf Basis von Skriptsprachen, wie zum Beispiel PHP und Perl, entwickelt wurden. Lediglich zwei der verglichenen Shop-Systeme beruhen auf Java, einer reinen objektorientierten Programmiersprache. Außerdem wurde festgestellt, dass oft SQL Datenbanken, besonders die weitverbreitete Datenbank MySQL, zur Datenhaltung und -sicherung genutzt wird. Gründe hierfür sind die erwähnte Sicherung der Daten, sowie die Leistungsfähigkeit und die Verwendung von Datenbankclustern.

Unterschiede zwischen den einzelnen Shop-Systemen konnten unter Anderem im Punkt Skalierbarkeit und verteiltes System festgestellt werden. Hier zeichnet sich die Tendenz ab, dass nur wirklich die Lösungen für große Unternehmen und multinationale Konzerne über eine variable Anpassbarkeit verfügen und verteilt auf mehreren Systemen laufen können. Dies bedeutet, dass diese Systeme in der Lage sind, viele Anfragen an den Shop und große Datenmengen leistungsfähig und effizient zu bewältigen.
Eine weitere Differenz wurde im Punkt Multisite ermittelt. Es zeigte sich, dass es – abgesehen von den teuren Lösungen - keine Unterstützung für mehrere Shops im gleichen Shop-System gibt. Es lässt den Schluss zu, dass dies mit der Skalierbarkeit der Systeme zusammenhängen könnte.

[16] http://www.magentocommerce.com/de/
[17] http://ofbiz.apache.org/
[18] http://www.oxid-esales.com/de/startseite
[19] http://www.1und1.info/xml/order/Eshops

## 2.4 Schlussfolgerung

Anhand, der zuvor durchgeführten Auswertung, ließen sich folgende Schlussfolgerungen ziehen.
Zum Einen hat sich gezeigt, dass die ausgewählten Shop-Systeme ein sehr weites Spektrum abdecken, was die eigentliche Auswahl der Systeme positiv bestätigt. Desweiteren konnte festgestellt werden, dass die Shop-Systeme an Flexibilität, Erweiterbarkeit, Leistungsfähigkeit und Skalierung zunehmen, angefangen von kleinen Systemen bis zu den großen Systemen. Also gibt es einen deutlichen Zuwachs von den Shop-System-Lösungen für private Personen und Kleinstunternehmen zu den Lösungen für große Unternehmen und multinationale Konzerne.
Diesbezüglich wurde festgestellt, dass die verglichenen Systeme zwar über einen gewissen Grad der Flexibilität und der Erweiterbarkeit verfügen, aber irgendwann an ihre Grenzen stoßen. Die Shop-Systeme sind hauptsächlich in die vertikale Richtung nach oben hin mehr, oder weniger offen, beziehungsweise erweiterbar und abänderbar. Sie lassen keine Änderungen der eigentlichen Geschäftslogik zu. Das heißt, die Systeme sind nicht in der horizontalen Richtung, nach rechts und links, veränderbar. Sie profitieren hauptsächlich von der vorhanden Basis und den, teilweise zahlreich, vorhanden Grundfunktionen.

Außerdem wurde ermittelt, dass vorhandene Shop-Lösungen nur begrenzt vergrößert und verkleinert werden können. Gemeint ist damit, wenn man ein vorhandenes Shop-System hat und dieses auf ein leistungsfähigeres System vom gleichen Hersteller wechselt, dies gar nicht oder nur begrenzt möglich ist und der Migrationsprozess relativ viel Aufwand bedeutet. Genauso verhält es sich, wenn man ein vorhandenes Shop-System verkleinern möchte.
Es konnte auch festgestellt werden, dass einige Shop-Systeme, besonders die kleineren, gar keine Option bieten, sich zu verkleinern oder zu vergrößern. Hier definiert die verwendete Hardware und Software die Grenzen des Shops.

Generell hat sich gezeigt, dass es keine Shop-Lösungen gibt, die gleichermaßen sinnvoll für private Personen und Kleinstunternehmen sowie für KMU, Großunternehmen und multinationale Konzerne eingesetzt werden. Jede Lösung deckt nur einen gewissen Bereich ab. Die eine Shop-Lösung hat eine größere Spanne, die andere Shop-Lösung hat eine kleinere Spanne.
Desweiteren weißt die Auswertung daraufhin, dass die meisten der verglichenen Shop-Systeme zwar modular aufgebaut sind, aber diese nicht die notwendigen Bedingungen erfüllen, um wirklich von einem modularen Softwaresystem sprechen zu können.

# 3 Technologie-Entscheidung

Im vorherigen Kapitel 2 Shop-System-Überblick wurde aufgezeigt, dass die untersuchten Shop-Systeme nicht alle relevanten Anforderungen erfüllen, welches ein modulares Softwaresystem auszeichnet. Deswegen ist es notwendig, eine geeignete Technologie zu finden, mit der man einen Prototyp entwickeln kann, welcher diesem System gerecht wird.

Als ersten Ansatz wurden die verwendeten Technologien in Betracht gezogen, die die untersuchten Shop-Systeme verwenden. Dies sind PHP, Perl und Java. Die beiden ersten genannten Technologien PHP und Perl sind Skriptsprachen. Diese sind von ihrer Architektur her, eher ungeeignet. Sie ermöglichen es zwar modularisiert ein Programm, Applikation usw. zu entwickeln, ihnen fehlen aber die Mechanismen um diese einzelnen Module, beziehungsweise die Kernfunktionalitäten, statisch und dynamisch zu verändern oder auszutauschen. Dies wird aber für ein modulares Softwaresystem benötigt. Somit müssten diese Mechanismen überhaupt erst entwickelt werden. Damit wäre zwar eine Entwicklung eines Prototyps möglich, aber es würde einen großen Aufwand bedeuten, die relevanten Mechanismen zu entwickeln und umzusetzen.
Zusätzlich zu den beiden Skriptsprachen wurde Java in die Technologieauswahl mit einbezogen. Desweiteren wurde C# und C++ mit einbezogen, weil diese, wie Java, objektorientierte Programmiersprachen sind. Allerdings stellte sich auch hier heraus, dass diese zwar besser geeignet wären, aber die entsprechenden Frameworks fehlen oder nicht so verbreitet sind, um ein modulares Softwaresystem zu entwickeln. Genau wie bei den beiden Skriptsprachen PHP und Perl, müssten erst die notwendigen Mechanismen entwickelt werden.

Aus diesem Grunde wurde nach einem Framework gesucht, welches diese Mechanismen bereits besitzt. Deswegen ist die Wahl auf das OSGi Framework gefallen. Wie im Kapitel 1.3 OSGi schon erläutert wurde, besitzt das OSGi-Framework diese Mechanismen. Es ist mögliche ein System in seine kleinsten Teile aufzuspalten, also Module. Diese Module entsprechen einem OSGi-Bundle. Desweiteren ist die Bedingung erfüllt, dass diese kleinsten Einheiten unabhängig existieren können. Der Verfasser dieser Bachelorarbeit verweist auf den Punkt 1.3.1 Architektur, Abschnitt Lebenszyklus-Schicht, welcher den Lebenszyklus eines OSGi-Bundle beschreibt. Der Lebenszyklus eines Bundle ermöglicht es, dass ein Bundle zur Laufzeit eingeklinkt werden kann, ausgewechselt werden kann, durch ein anderes Bundle ersetzt und entfernt werden kann. Dies bedeutet auch, dass ein Service sich, zur Laufzeit, verändern kann. Dadurch ist es möglich, das, in Kapitel 1.1 Modulares Softwaresystem beschriebene, **Pluggable-Konzept** umzusetzen. Zudem ist es möglich für

Module, also Bundles, einheitliche Schnittstellen zu definieren. Dies wird im OSGi-Framework durch die OSGi-Services umgesetzt. Ein weiterer, wichtiger Aspekt für ein modulares Softwaresystem ist die Abgrenzung der eigentlichen Implementierung eines Moduls. Dies wird mit der Verwendung von Service-Interfaces im OSGi-Framework umgesetzt.
Durch die Angabe in der Manifest-Datei in einem Bundle ist es außerdem möglich, interne Bundle-Ressourcen für andere Bundle bereitzustellen.

# 4 Entwicklung

In diesem Kapitel wird die allgemeine Herangehensweise beschrieben, wie bei der Entwicklung des Prototyps für ein Shop-System auf Basis von OSGi-Komponenten vorgegangen wurde und warum. Zusätzlich wird die Auswahl weiterer Technologien erläutert.

## 4.1 Grundkonzept

Als erstes wurde überlegt, welche grundlegenden Komponenten der Prototyp benötigt beziehungsweise aus welchen wesentlichen Komponenten ein Shop-System besteht. Aufbauend auf den theoretischen Grundlagen, siehe Punkt 1.2.1 Aufbau - eines Shop-Systems, wurden folgende Komponenten ermittelt. Die folgende Abbildung wurde zur Veranschaulichung dieser Komponenten erstellt.

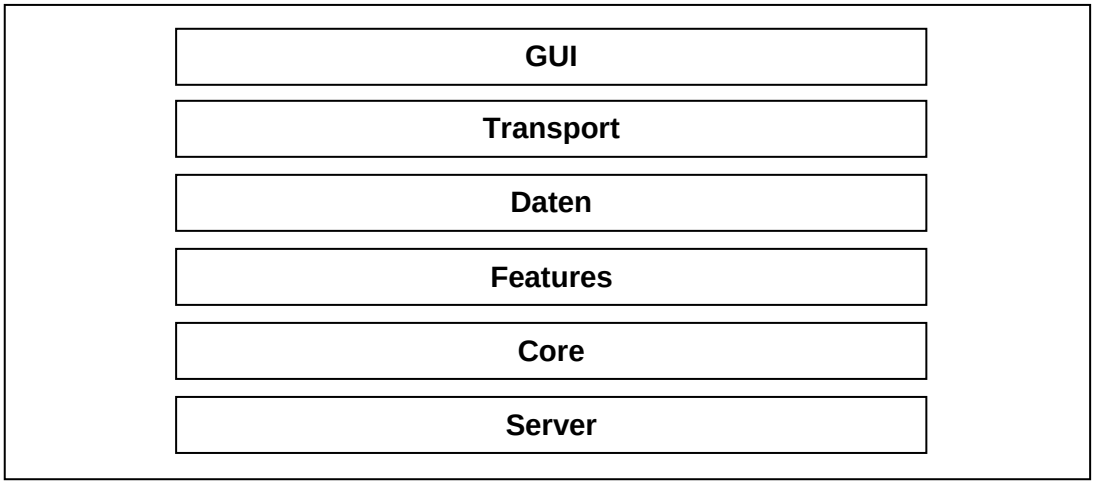



**Abbildung 7: Grober Systemüberblick**

Wie zu sehen ist, werden sechs Komponenten dargestellt, in der Abbildung 7. Die Bedeutung dieser Komponenten wird nachfolgend näher erläutert.

Um die Bedienung des Prototyps durch den User beziehungsweise durch den Benutzer zu gewährleisten besser gesagt überhaupt erst zu ermöglichen, ist eine GUI-Komponente das heißt eine grafische Oberfläche, erforderlich. Übertragen auf ein Shop-System bedeutet dies, dass ein Front-End als eine grafische Oberfläche benötigt wird, die es den Besuchern erlaubt, den Shop zu benutzen. Zum Beispiel: Produktkategorien auszuwählen, sich Produkte anzeigen zu lassen, eine Bestellung von ausgewählten Produkten vorzunehmen, kurz gesagt – im Shop einzukaufen. Zusätzlich wäre die Entwicklung eines grafischen Back-

Ends denkbar. Zum Beispiel könnten im Backend Shop-Einstellungen vorgenommen werden, Produkte dem Shop-System neu hinzugefügt, gelöscht und geändert werden. Es muss also eine grafische Oberfläche entwickelt werden, die als Schnittstelle zwischen dem Benutzer und dem Shop-System fungiert. Es sei angemerkt, dass das Front-End die Schnittstelle zum eigentlichen Endkunden ist, also zu jenen Benutzern, die im Onlineshop einkaufen, oder sich die angebotenen Artikel ansehen wollen. Wohingegen das Back-End die Schnittstelle für den Betreiber des Onlineshops darstellt.

Im vorherigen Abschnitt wurde die Notwendigkeit einer grafischen Oberfläche beschrieben. Deshalb wurde darauf geschlossen, dass der Prototyp eine Möglichkeit benötigt, um mit dem Shop-System zu kommunizieren, entweder direkt oder indirekt. Zum Beispiel, wenn der Benutzer den Onlineshop in seinem Internetbrowser aufruft, werden die erforderlichen Daten für die Anzeige des Onlineshops vom Shop-System an den Internetbrowser des Benutzers übertragen. Dies ist allerdings stark vereinfacht beschrieben.

Durch die Eingaben des Benutzers in seinem Interbrowser werden die Daten nicht direkt an das Shop-System geschickt. Vielmehr ist es so, dass die Daten an den Teil des Shop-Systems geschickt werden, der dafür verantwortlich ist, dass das Front-End im Interbrowser des Benutzers dargestellt wird. Das Front-End ist also eine Webapplikation.

*„Eine Web-Applikation ist ein meist datenbankgestütztes Computerprogramm, das zu einem wesentlichen Teil auf einem Webserver läuft und mittels eines Browsers per Intranet/Internet von einem Anwender benutzt und bedient wird.“* [20]

Dies bedeutet, eine Webapplikation kann im Interbrowser des Benutzer ausgeführt werden, muss also nicht auf dem PC des Benutzers installiert werden. Dafür ist auf Seiten des Shop-Systems ein Webserver erforderlich, auf dem die Webapplikation, in unserem Fall das Frontend, läuft. Damit stellt die Komponente Server, wie der Name andeutet, die notwendigen Server-Komponenten dar, die für den Prototyp benötigt werden. Mit dem Begriff des Servers ist aber nicht die Definition eines Hardware-Servers gemeint, also einem physikalischen Server. Vielmehr ist der Begriff des Servers, im Zusammenhang mit Software, gemeint. Das heißt, mit Server ist ein Programm gemeint, welches auf einem physikalischen Server installiert ist und einen Dienst anbietet. Das Programm reagiert auf die Anfrage des Benutzers und beantwortet diese Anfrage. Die Kommunikation zwischen Server und Client beziehungsweise zwischen Server und Benutzer ist Bestandteil des sogenannten Client-Server-Modells. Dieses Modell soll lediglich hier Erwähnung finden und wird nicht

[20] Zitiert von ITos GmbH (2011), 1. Abs.

näher erklärt, weil dies sonst den Umfang der Bachelorarbeit zu weit ausdehnen würde. Um es schon einmal vorweg zu nehmen, der Prototyp benötigt nicht nur einen Webserver, sondern auch einen Datenbankserver.

Da es sich um einen Onlineshop handelt, ist es erforderlich, benötigte Daten konsistent zu speichern. Ein Beispiel ist hierfür, das Speichern von Produktdaten oder von Kundendaten. Mit der Komponente Daten ist gemeint, dass Daten, die für das Shop-System beziehungsweise den Prototyp erforderlich sind, in einer gewissen Form und Art und Weise gespeichert werden. Das heißt, zum dauerhaften und konsistenten Speichern von Daten wird eine Datenbank genutzt respektive ein Datenbankserver. Der Grund für die Verwendung eines Datenbankservers wird in einem der nachfolgenden Punkte erläutert.

Außerdem wurde angedacht, das Shop-System mit einem Kern, engl. Core, auszustatten. In diesem Kern sollen Grundfunktionen für das System vorhanden sein, wie zum Beispiel Zugriff auf die Datenbank beziehungswweise den Datenbankserver und diverse Grundfunktionalitäten, beispielsweise das Logging. Mit dem Begriff Logging ist das Protokollieren von Ereignissen gemeint. Neben der gerade beschriebenen Core-Komponente wurde vom Verfasser dieser Bachelorarbeit zusätzlich eine Features-Komponente angedacht. Diese beschreibt im Wesentlichen die verschiedenen Funktionalitäten des Shop-Systems. Der Grund hierfür ist, dass die Core-Komponente lediglich die grundsätzlichsten Funktionalitäten des Shop-Systems mit sich bringt und damit sozusagen nur eine Minimalausstattung bietet.

## 4.2 Auswahl von Technologien

In den nachfolgenden Abschnitten wird unter Anderem beschrieben, für welche Implementierung eines OSGi-Frameworks sich entschieden wurde. Außerdem wurde nach der Ermittlung der Komponenten für den Prototyp des Shop-Systems festgestellt, dass noch weitere Technologien und externe Java Bibliotheken für die Umsetzung erforderlich sind. Nachfolgend wird beschrieben, warum und welche zusätzlichen Technologien ausgewählt wurden.

### OSGi-Implementierung

Für die Entwicklung des Prototyps wurde sich für Equinox[21] als OSGi-Framework-Implementierung entschieden. Der Grund dafür lag darin, dass der Verfasser dieser Bachelorarbeit mit der integrierten Entwicklungsumgebung Eclipse[22] bereits zahlreiche Erfahrungen gemacht hat. Der eigentliche und ausschlaggebende Grund war, dass Eclipse selbst seit der Version 3 auf der hauseigenen OSGi-Implementierung Equinox aufbaut. Dadurch bietet das Entwicklungstool Eclipse zahlreiche Mechanismen und Werkzeuge, welche die Entwicklung von OSGi-Anwendungen erleichtern.

### Logging

Im vorherigen Punkt 4.1 Grundkonzept wurde bereits kurz erwähnt, dass für den Prototyp ein Log-Mechanismus verwendet werden soll. Dieser Log-Mechanismus soll dazu dienen, dass Ereignisse, besser gesagt Log-Events, protokolliert werden, zum Beispiel in einer Log-Datei. Für den Prototyp wurde angedacht, nur die serverseitigen Ereignisse aufzuzeichnen, da dies in der Praxis gängig ist. Es wurde eine Auswahl getroffen aus den gängigsten und bekanntesten Log-Frameworks für Java. Diese waren in diesem Fall SLF4J[23], Apache log4J[24] und der Java Logging API[25], welche standardmäßig, unter Anderem im Java Development Kit seit Version 1.4, enthalten ist.
Der Verfasser dieser Bachelorarbeit hat sich für das Log-Framework log4j von Apache entschieden, da dieses einfach zu benutzen ist und darüber hinaus die Möglichkeit bietet, die aufgezeichneten Log-Events auf mehrere Dateien aufzuteilen.

[21] http://www.eclipse.org/equinox/
[22] http://www.eclipse.org/
[23] http://www.slf4j.org/
[24] http://logging.apache.org/log4j/
[25] http://download.oracle.com/javase/1.4.2/docs/guide/util/logging/

**Daten**

Für die Datenhaltung, zum Beispiel die Speicherung von Produktdaten, Katalogdaten und Bestelldaten, wurde sich für eine datenbankgestützte Lösung entschieden. Denauer gesagt wurde sich für die Open Source Lösung MySQL entschieden. Die Gründe hierfür waren die Leistungsfähigkeit, konsistente Datenhaltung, Unterstützung eines relationalen Datenbankmodells, gute Dokumentation, kostenlos und die bisherigen Erfahrungen des Autors dieser Bachelorarbeit mit SQL-Datenbanken.

**GUI**

Da das Front-End des Shop-Systems die Schnittstelle zum Benutzer beziehungsweise zum Kunden hin darstellt, muss eine Technologie gefunden werden, mit der man sowohl eine grafische Darstellung im Browser des Benutzers realisieren kann, als auch die Kommunikation mit dem Shop-System beziehungsweise dem Prototyp ermöglicht. Gemeint ist also, dass nach einer Technologie gesucht wird, mit deren Hilfe man eine Webapplikation entwickeln kann. Darüber hinaus wäre es wünschenswert, wenn die Webapplikation aus dem OSGi-Container heraus lauffähig wäre, beziehungsweise wäre es von Vorteil, wenn die Webapplikation als OSGi-Bundle in das Shop-System eingebunden werden könnte und darüber hinaus dynamisch mithilfe eines OSGi-Services vom Benutzer beziehungsweise vom Kunden genutzt werden könnte. Zusätzlich zu den bisherigen Kriterien sollte es möglich sein, dass dynamische Webtechnologien, wie zum Beispiel JavaScript und CSS, verwendet werden. Gründe dafür sind Flexibilität des Front-Ends und der wichtige Aspekt des Customizing. Damit ist speziell das Ändern des Layouts der Webapplikation gemeint.

Anhand, der bisher beschriebenen Anforderungen wurde GWT[26] als passende Technologie, für die Front-End-Webapplikation ausgewählt. Nachfolgend wird die Funktionsweise beschrieben und was GWT ist. Mithilfe von GWT ist es möglich, browserbasierte AJAX-Webapplikationen zu entwickeln. Zudem bietet das Google Web Toolkit Cross-Browser-Support, die Unterstützung für Mehrsprachigkeit, HTML 5 Unterstützung ab Version 2.3, diverse Sicherheitsmechanismen, eine gute Dokumentation und die Möglichkeit für JUnit-Tests.[27] Nachfolgend wird das Web Toolkit von Google kurz beschrieben. Durch GWT ist es möglich, dass Java-Entwickler komplexe und moderne Webapplikationen entwickeln können, ohne zwingend Kenntnisse in JavaScript und CSS zu besitzen. Der geschriebene Java-Quellcode wird mittels des mitgelieferten Compilers in eine Webanwendung kompiliert. Das heißt der clientseitig geschriebene Java-Quellcode wird in eine JavaScript-Anwendung

[26] http://code.google.com/intl/de-DE/webtoolkit/
[27] Quelle: Google (2011)

kompiliert. Diese wird dann mit den Webtechnologien CSS und HTML kombiniert. Dadurch ist es möglich, dass komplett dynamische Inhalte erzeugt werden können. Aber auch die Kombination mit statischen Inhalten ist dadurch möglich. Die Kommunikation zwischen dem serverseitigen Code und dem clientseitigen Code erfolgt mittels GWT RPC.[28]

**Server**

Um die im vorherigen Abschnitt beschriebene Webapplikation ausführen zu können, bedarf es eines Webapplikation-Servers. Oberstes Ziel war es, einen Webapplikations-Server zu finden, der eingebettet in einem OSGi-Bundle läuft und der die Möglichkeit bietet, dynamische und statische Webinhalte programmatisch einzubinden. Daraufhin wurde sich eine Übersicht über vorhandene Lösungen für Webapplikations-Server verschafft. Im Anschluss wurde die Auswahl auf zwei mögliche Varianten eingegrenzt. Diese Varianten sind die eingebetteten Versionen von Apache TomCat[29] und Jetty[30]. Beide Varianten sind Open Source und vollwertige Webserver und unterstützen zu dem die Verwendung von Java-Servlets, welche unteranderem von GWT in einer erweiterten Form genutzt werden. Der Verfasser dieser Bachelorarbeit hat sich aus folgenden Gründen für die Verwendung von Embedded Jetty entschieden. Er ist leicht zu bedienen, programmatisch oder über eine XML-Datei konfigurierbar, leistungsfähig und kostenlos. Desweiteren besitzt Jetty seit der Version 7 offiziell die Unterstützung für das von GWT verwendete AsyncRemoteServiceServlet, das die standardmäßige Servlet-Defintion erweitert.

[28] Quelle: STEYTER, RALPH (2007), Google (2011)
[29] http://tomcat.apache.org/
[30] http://www.eclipse.org/jetty/

## 4.3 Detailliertes Konzept

Nach der Entwicklung eines Grobkonzeptes und der Auswahl geeigneter Technologien wurde das Konzept weiter verfeinert.

### 4.3.1 Technischer Systemüberblick

Zuerst wurde ein technischer Systemüberblick erarbeitet. Dieser soll aufzeigen wie die einzelnen Bestandteile des Prototyps zusammenarbeiten. Nachstehende Abbildung visualisiert die Abhängigkeiten zwischen den einzelnen Bestandteilen.

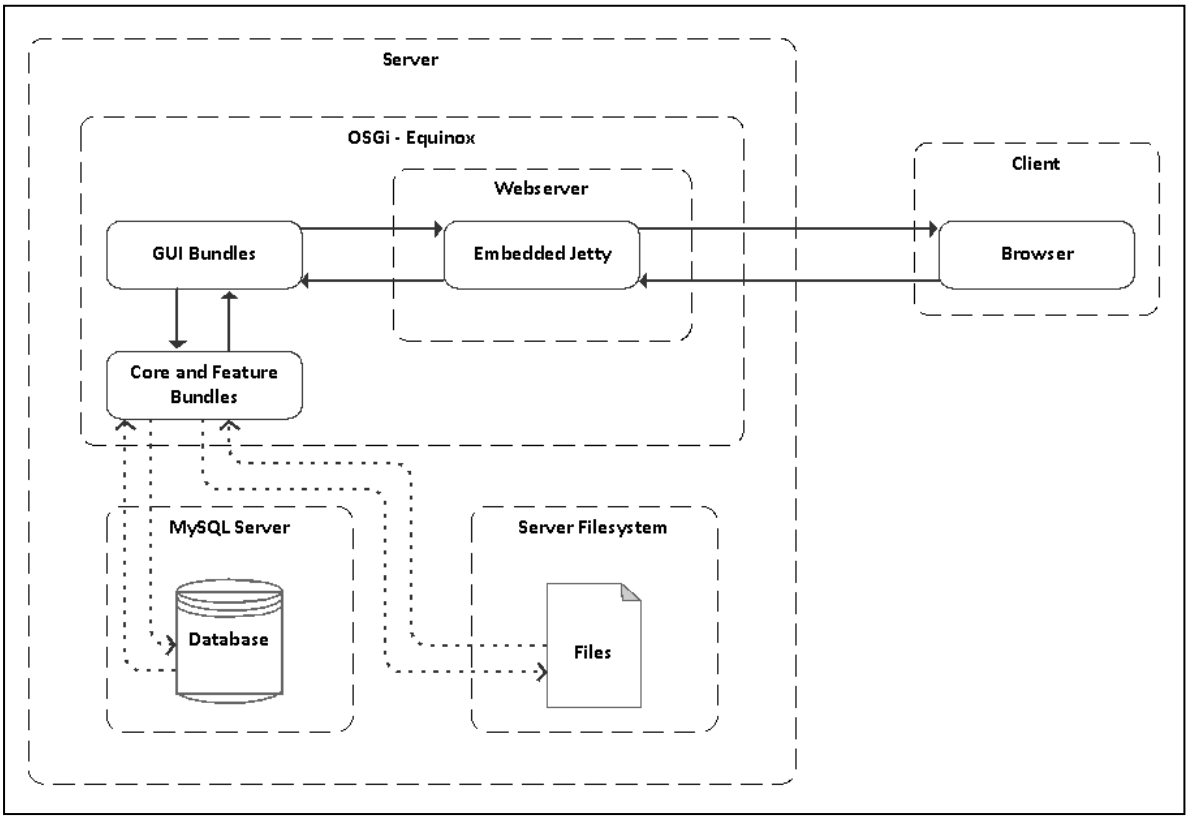



**Abbildung 8: Technischer Systemüberblick**

Wie in Abbildung 8 zu sehen ist, liegen alle Bestandteile des zu entwickelnden Shop-System-Prototyps auf einem einzigen Server. Dies hat folgende Gründe. Zum einem sollte die grafische Darstellung übersichtlich gehalten werden, zum anderen ist es für Entwicklung des Prototypen nicht notwendig, ein verteiltes System beziehungsweise mehrere Hardware-Server zu nutzen.

Nachfolgend wird die Funktionsweise anhand eines Beispiels erklärt. Der Benutzer, der den Client repräsentiert, ruft über den Browser die Startseite des Prototyp-Front-Ends auf. Damit stellt er eine Anfrage an den Webserver, in dem Fall eine eingebettete Jetty Version, welche

als Bundle im OSGi-Container läuft. Im Vorfeld hat sich das GUI-Bundle, welches das Front-End des Prototyps beinhaltet, schon per OSGi-Service beim Jetty-Server registriert. Ausgehend von der ersten Anfrage des Clients an den Server finden automatisch weitere Anfragen statt. Der entscheidende Punkt ist, dass erst der clientseitige Code übertragen wird, welche die erste Antwort vom Server in Bezug auf die erste gestellte Anfrage vom Client darstellt. Daraufhin finden, ausgehend vom clientseitigen Code, weitere Anfragen an den Server beziehungsweise an den serverseitigen Code der Front-End-Webapplikation statt. Der serverseitige Code nutzt OSGi-Services des Core-Bundle und der Feature-Bundle, zum Beispiel einen Log-Service, einen Katalog-Service, einen Datenbank-Service usw. Diese Core- und Feature-Bundle können Zugriff auf den lokalen Datenbankserver und das Dateisystem vom Server haben. Das wäre zum Beispiel der Fall, wenn ein OSGi-Bundle einen Service anbietet, der Produktdaten vom Datenbankserver holt oder Kundendaten auf den Datenbankserver speichert. Ein anderes Beispiel wäre, wenn ein OSGi-Bundle einen Service anbietet, der den Zugriff auf Konfigurationsdateien ermöglicht, die auf dem lokalen Dateisystem vom Server liegen.

### 4.3.2 Struktureller Systemüberblick

Aufbauend auf den in Punkt 4.1 Grundkonzept beschrieben Komponenten und der in Punkt 4.2 ausgewählten Technologien wurde die Komponenten-Darstellung des Shop-System-Prototyps weiter spezifiziert.

Die Komponente GUI teilt sich in zwei Bestandteile auf. Zum einen GWT, welches für die grafische Darstellung, also das Front-End, zuständig ist. Zusätzlich wurde die mögliche Verwendung von Java-Servlets angedacht. Wie schon erwähnt, nutzt GWT eine erweiterte Form der Servlet-Definition. Darauf aufbauend ergeben sich für die Komponente Transfer ebenfalls zwei Bestandteile. Zum einen HTTP als Übertragungsprotokoll, zum anderen die von GWT verwendete RPC-Technologie, welche grundlegend auf den verbindungslosen Transport UDP aufsetzt und HTTP als Transportprotokoll nutzt.

Die drei Server-Komponenten sind der eingebettete Webapplikation-Server Jetty, der MySQL-Datenbankserver und der OSGi-Equinox-Server. Als Daten-Komponenten wurde die Datenbank MySQL und Properties-Files spezifiziert. Die MySQL-Datenbank ist zuständig für die konsistente Datenhaltung und das Datenmodell. Die Properties-Files liegen auf dem lokalen Dateisystem des Servers und sollen die OSGi-Bundle konfigurierbar machen.

Die Komponente Core besteht aus acht Bestandteilen: Properties, Logging, Paths, Model, Email, Order, Catalog und Database. Jeder dieser einzelnen Bestandteile soll dazu dienen, den Kern des Shop-System-Prototypen mit wichtigen Grundfunktionalitäten auszustatten. Der Verfasser dieser Bachelorarbeit verzichtet an dieser Stelle auf eine detaillierte Beschreibung der einzelnen Bestandteile, da die eigentliche Funktion größtenteils aus den Namen hervorgeht.

Für die Komponente Feature werden nachfolgend mögliche Beispiele genannt. Dies hat den Grund, dass die Features zwar die eigentlichen Funktionalitäten des Shop-Systems ausmachen, aber für den Prototyp so keine Anwendung finden. Möglicher Bestandteil der Feature-Komponente wäre zum Beispiel ein Bestandteil Statistik, welcher User-Statistiken aufzeichnet, oder ein Bestandteil PayPal, welcher die Zahlung einer Bestellung per PayPal ermöglicht. Zur Veranschaulichung der beschrieben Komponenten und deren Bestandteile wurde die folgende Grafik angefertigt.

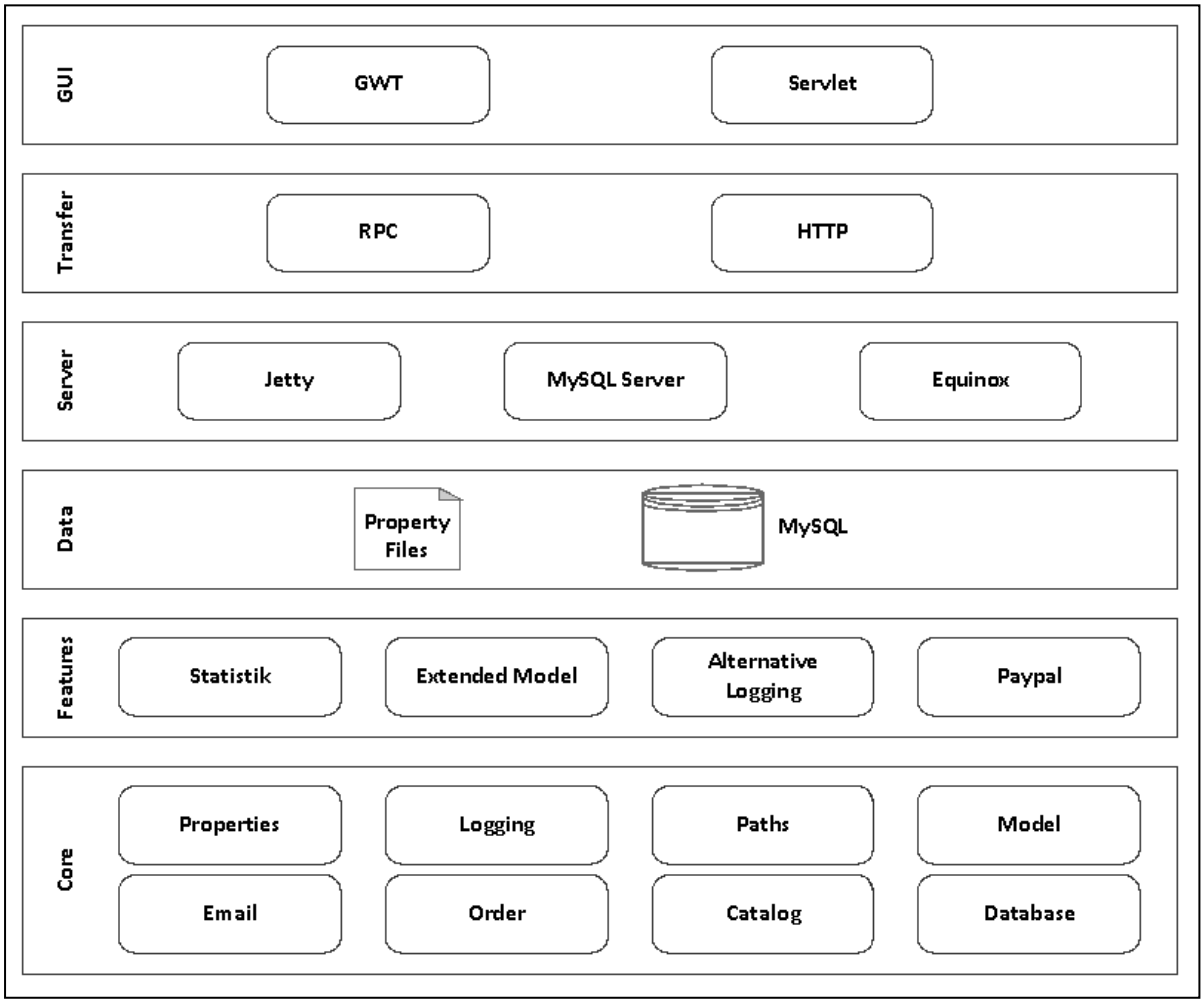



**Abbildung 9: Struktureller Systemüberblick**

### 4.3.3 Datenbankmodell

Für den Prototyp des Shop-Systems wurde außerdem ein Datenbankschema entworfen. Dieses beschreibt im Wesentlichen wie die Katalogdaten, die Produktdaten, Zahlungsdaten, Versanddaten und Kundendaten in der Datenbank abgespeichert werden. Die folgende Abbildung stellt dies grafisch dar.

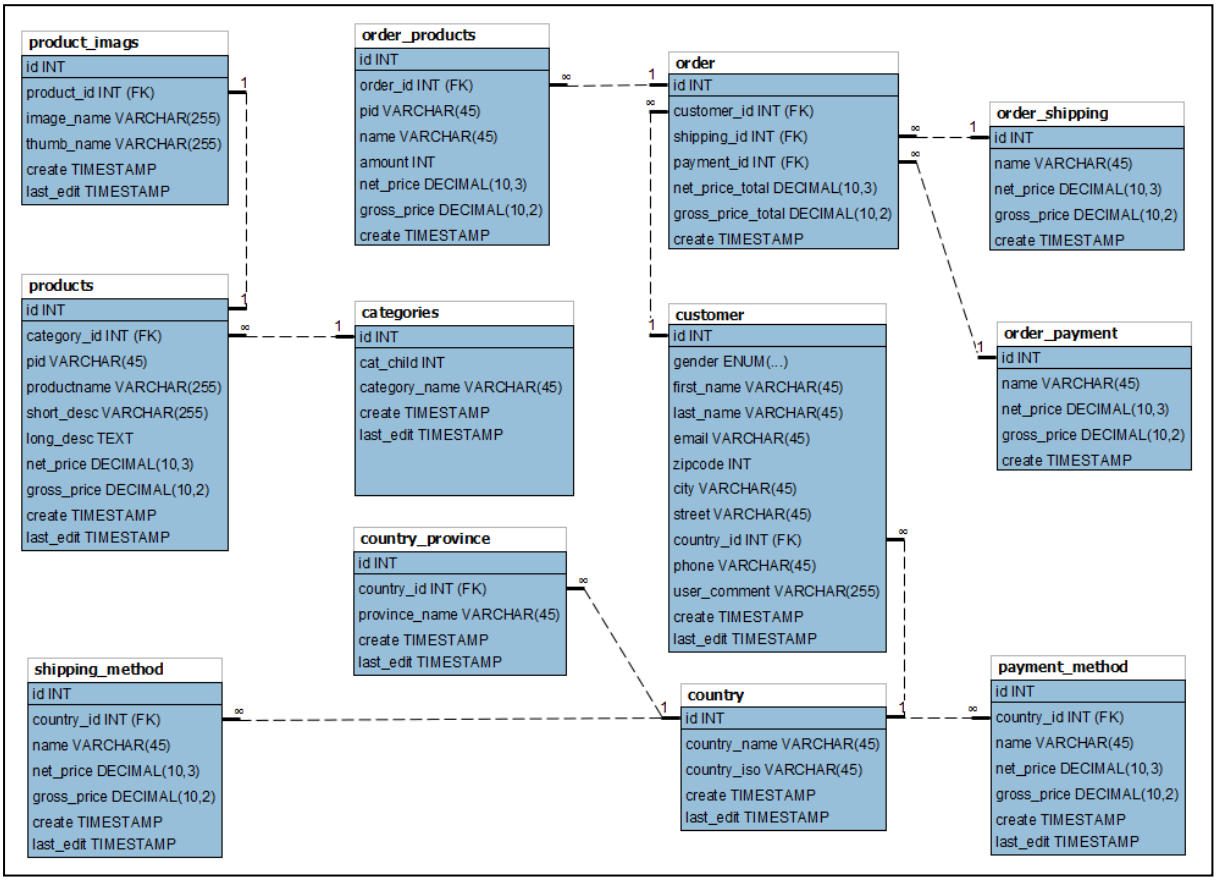



**Abbildung 10: Datenbankmodell**

Aufgrund des Umfangs dieser Bachelorarbeit hat sich der Verfasser dazu entschieden, auf die Beschreibung des Datenbankmodells zu verzichten.

### 4.3.4 Prozesse

Des Weiteren wurden für das Front-End und für das grafische Back-End die Kernprozesse spezifiziert. Die Abbildung 11 und die Abbildung 12 stellen dies grafisch dar.

**Front-End**

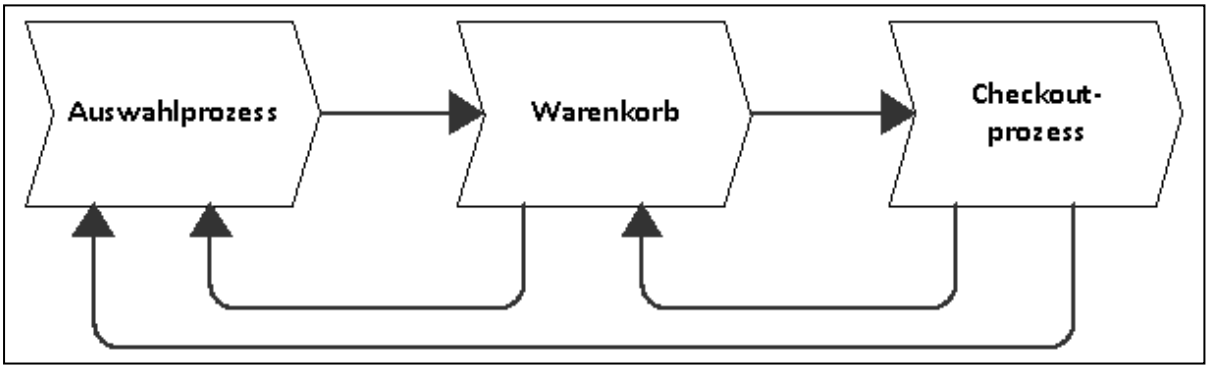



**Abbildung 11: Front-End Prozessübersicht**

Für das Front-End wurden drei Kernprozesse spezifiziert. Der Auswahlprozess, welcher für die eigentliche Produktauswahl steht. Der Prozess Warenkorb steht für die Übersicht der bisher ausgewählten Produktartikel. Und der Checkoutprozess, welcher für die Abwicklung der eigentlichen Bestellung steht. Damit ist gemeint: Die Angabe der Kundendaten, Auswahl der Bestelldetails, wie zum Beispiel Zahlungsart und Versandart, Anzeige der gesamten Bestellung, inklusive Gesamtbetrag der Bestellung und das Anzeigen der Auftragsbestätigung. Für die drei Kernprozesse wurde jeweils ein Anwendungsfalldiagramm entwickelt. Diese sind im Anhang auf Seite XII ff. zu finden.

**Back-End**

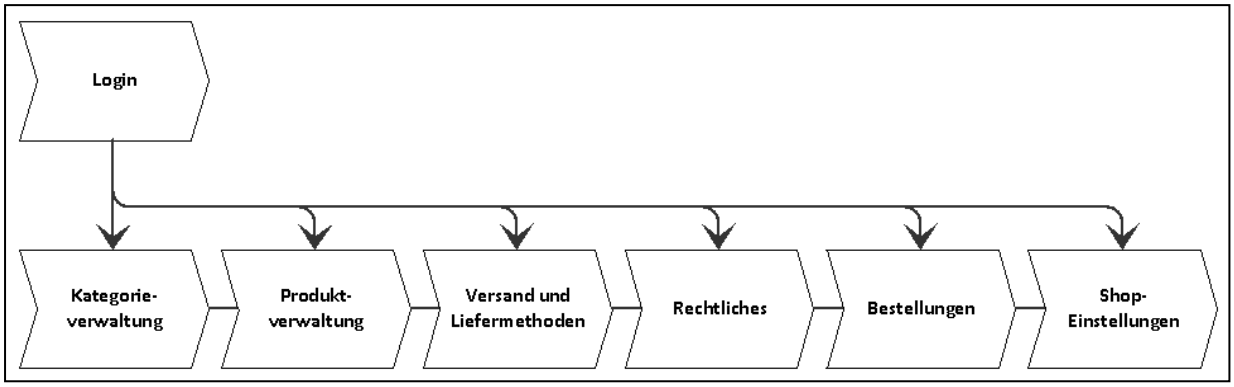



**Abbildung 12: Back-End Prozessübersicht**

Da in vielen Shop-Systemen die Verwendung eines grafischen Back-Ends üblich ist, wurde angedacht, ebenfalls eine grafische Back-End Umsetzung zu entwickeln. Aus diesem Grund wurden die in der oberen Abbildung 12 dargestellten Prozesse spezifiziert.
Es wurde angedacht, dass es einen Login-Prozess gibt, wo sich der Benutzer am Back-End anmelden kann. Wie in der oberen Abbildung 12 zu erkennen ist, kann vom Login-Prozess ausgehend - jeder andere dargestellte Prozess erreicht werden. Zudem ist es angedacht, dass zwischen den einzelnen Prozessen für das Back-End frei gewechselt werden kann, ausgenommen davon ist der Login-Prozess. Es gibt also keine starre Prozessreihenfolge.
Für die Kernprozesse des Back-Ends wurden, wie schon beim Front-End, jeweils ein Anwendungsfalldiagramm entwickelt, welche sich im Anhang auf Seite XV befinden.

## 4.4 Implementierung

In diesem Punkt wird nachfolgend die allgemeine Herangehensweise bei der Implementierung sowie aufgetretene Probleme, Hindernisse und die Aufgaben der einzelnen Bundle, beschrieben. Außerdem werden wichtige Punkte mit Quellcode-Beispielen belegt.

Vor dem Beginn der eigentlichen Implementierung wurde der lokale MySQL-Server installiert. Danach wurde die Datenbank für den Prototyp erstellt und die Datenbanktabellen angelegt. Grundlage hierfür war das selbstentwickelte Datenbankschema, siehe Punkt 4.3.3 Datenbankmodell. Zudem wurden die Datenbanktabellen mit Testdatensätzen für die eigentliche Implementierung gefüllt.

**Properties-Bundle**

Im ersten Schritt wurde die Implementierung des Properties-Bundle vorgenommen. Dieses Bundle bietet einen Service an, der sich für andere Bundle darum kümmert, wie und wo sie ihre Eigenschafte, also Properties, abspeichern und laden können. Das Service-Interface für diesen Properties-Service gestaltet sich daher recht einfach. Es wurden 2 Methoden für das Service-Interface definiert, welche das Laden und Speichern einer Eigenschaft ermöglichen. Zudem muss jeweils, bei beiden Methoden der Bundle-Name angegeben werden. Der Grund liegt darin, dass sich dafür entschieden wurde, für jedes Bundle eine eigene Properties-Datei zu nutzen.

```
01  import java.util.Properties;
02  public interface IPropertiesService {
03        public Properties loadProperties(String symbolicBundleName);
04        public boolean saveProperties(Properties properties,
05                    String symbolicBundleName);
06  }
```

**Listing 8: Properties-Service-Interface**

Wie zu sehen ist in Listing 8, ist das die Interface-Klasse recht einfach definiert. Die loadProperties()-Methode erfordert einen Bundle-Namen, in Form eines Java-String-Objektes und liefert ein Java-Properties-Objekt zurück. Die saveProperties()-Methode definiert zwei Methoden-Variablen, die mit übergeben werden müssen: Zum Einen den Bundle Namen, in Form eines Java-String-Objektes und zum Anderen ein Java-Properties-Objekt.

Neben dem Service-Interface befinden sich im Properties-Bundle zwei weitere Java-Klassen. Zum Einen, eine Activator-Klasse, welche den Properties-Service in der OSGi-Registry anmeldet und zum Anderen die eigentliche Implementierung des Properies-Service-Interfaces, die Klasse PathServiceImpl, welche das Interface IPropertiesService implementiert, . Das Registrieren des Properties-Service sieht folgendermaßen aus.

```
01  import org.osgi.framework.BundleActivator;
02  import org.osgi.framework.BundleContext;
03  import de.igniti.osgi.shop.core.services.IPathService;
04  public class Activator implements BundleActivator {
05        private static BundleContext context;
06        static BundleContext getContext() {
07              return context;
08        }
09        public void start(BundleContext bundleContext) throws Exception {
10              Activator.context = bundleContext;
11              context.registerService(IPathService.class.getName(),
12                          new PathServiceImpl(), null);
13        }
14  … }
```

**Listing 9: Registrierung Properties-Service**

### Path-Bundle

An diesem Punkt wurde überlegt, einen Path-Service abweichend vom bisherigen Konzept zu implementieren. Dazu wurde ein neues OSGi-Bundle erstellt. Der Aufbau des Bundle gestaltet sich analog dem Properties-Bundle. Der Grund für das Path-Bundle liegt darin, dass es einen Service anbietet, den alle anderen Bundles nutzen sollen. Um konkreter zu werden, der Path-Service liefert den Pfad zu einem, in der eigentlichen Implementierung definierten Pfad. Dadurch soll für den Prototyp beziehungsweise die OSGi-Bundle ein

einheitliches Arbeitsverzeichnis erzielt werden. Davon profitiert zum Beispiel das Properties-Bundle, welches nun unter Verwendung des Path-Service einen eigenen Ordner für die Properties-Dateien nutzen kann. Dazu wurde der Properties-Service entsprechend angepasst.

**Logging-Bundle**

Das Logging-Bundle bietet einen Log-Service an, der es ermöglicht, dass andere Bundle diesen Service nutzen können, um bestimmte Ereignisse zu protokollieren. Hier wurde auch erkannt, dass die Service-Interfaces möglichst abstrakt und einfach spezifiziert sein sollten. Dabei ist es von Vorteil, wenn die Methoden der Service-Interfaces bekannte, also in der Java-Bibliothek schon vorhandene, Java-Datentypen verwenden. Deshalb wurde das Service-Interface so definiert, dass es nicht das Logger-Objekt selbst mit übergibt, sondern lediglich die Übergabe einer Log-Nachricht in Form eines Java-String-Objektes und die Übergabe eines Java-Class-Objektes erfordert. Das Log-Service-Interface wurde wie folgt spezifiziert.

```
01  public interface ILoggerService {
02        public void debug(String message, Class<?> clazz);
03        public void info(String message, Class<?> clazz);
04        public void warn(String message, Class<?> clazz);
05        public void fatal(String message, Class<?> clazz);
06        public void error(String message, Throwable e, Class<?> clazz);
07  }
```

**Listing 10: Log-Service-Interface**

Die Registrierung des Log-Service findet analog zum Path-Bundle und Properties-Bundle statt. Auch der Aufbau des Bundle ähnelt diesen.
In Punkt 4.2 Auswahl von Technologien wurde erläutert, dass Log4J als Logging-Framework verwendet wird. Die Konfiguration des Loggers findet über eine Konfigurationsdatei statt, welche im generellen Arbeitsordner liegt. Das Log-Bundle nutzt den Path-Service, um auf die Konfigurationsdatei Zugriff zu erhalten. Die Datei wurde so konfiguriert, dass die Log-Ausgaben sowohl auf der OSGi-Konsole sichtbar sind als auch in eine Log-Datei geschrieben werden. Zudem wird für jeden Tag ein einzelnes Log-Datum erstellt. Die erstellte Log-Konfigurationsdatei ist in Listing 11 einzusehen.

```
01  log4j.rootLogger=DEBUG, R, O
02  # Konsolen-Log
03  log4j.appender.O=org.apache.log4j.ConsoleAppender
04  log4j.appender.O.layout=org.apache.log4j.PatternLayout
05  log4j.appender.O.layout.ConversionPattern=[%d{ISO8601}] - %-5p [%t] %m%n
06  # Datei-Log
07  log4j.appender.R=org.apache.log4j.DailyRollingFileAppender
08  log4j.appender.R.datePattern='_'yyyy-MM-dd'.log'
09  log4j.appender.R.File=${log4j.logpath}
10  log4j.appender.R.layout=org.apache.log4j.PatternLayout
11  log4j.appender.R.layout.ConversionPattern=[%d{ISO8601}] - %-5p [%t] %m%n
```

**Listing 11: Beispiel einer Log4J-Konfigurationsdatei**

### Database-Bundle

Im nächsten Schritt wurde das Database-Bundle implementiert. Bei der Spezifizierung des Server-Interfaces wurde erst darüber nachgedacht, konkrete Methoden zu definieren, zum Beispiel für das Speichern eines Objektes oder das Lesen von Produktinformationen. Dieser Ansatz wurde aber verworfen, da jedes Bundle selbst definieren soll, wie es Informationen in die Datenbank schreibt und wieder ausliest. Deswegen wurde sich für eine einfache, aber flexible Definition des Database-Interfaces entschieden, welches das folgende Listing verdeutlicht.

```
01  import java.sql.Connection;
02  public interface IDatabaseService {
03        public Connection getConnection();
04  }
```

**Listing 12: Database-Service-Interface**

Das Java-Interface IDatabaseService definiert lediglich eine Methode getConnection(). Diese Methode liefert eine Java-SQL-Connection zurück. Vorteil dabei ist, dass das Übergebene Objekt ein Teil der Standard-Java-Bibliothek ist und damit jedes andere Bundle, welche den Databse-Service nutzt, das übergebene SQL-Connection-Objekt kennt und damit arbeiten kann. Außerdem können verschieden SQL-Anbindungen genutzt werden. In diesem Fall wurde in der konkreten Implementierung des Database Interfaces ein Java-Connector für eine MySQL-Datenbank verwandt. Der Aufbau des Database-Bundle und die Registirierung des Database-Service gestalten sich analog zu dem Aufbau des Properties- und Path-Bundle. Zudem nutzt das Database-Bundle den Properties-Service. Der Aufbau der Properties-Datei wird in Listing 13 dargestellt.

```
01  serverIP = localhost
02  serverPort = 3306
03  databaseName  = shop
04  databaseUser = root
05  databasePassword = a1b2c3
```

**Listing 13: Aufbau einer Properties-Datei**

### Catalog-Bundle

Das Catalog-Bundle bietet zwei Services an, zum Einen den Product-Service und zum Anderen den Category-Service. Dazu wurden ein Produkt-Model und ein Kategorie-Model erstellt. Im ersten Ansatz wurden diese beiden Klassen als Bean-Klasse implementiert. Das heißt, die beiden Model-Klassen besitzen einen Standard-Konstruktor sowie private Variablen. Auf diese privaten Variablen wird mit Hilfe von öffentlich Getter- und Setter-Methoden zurückgegriffen. Dies wird anhand der folgenden Catalog-Model-Klasse einmal verdeutlicht.

```
01  public class CatalogModel {
02        // Variablen
03        private int categoryID;
04        private int categoryChildID;
05        private String categoryName;
06        // Standardkonstruktor
07        public CatalogModel() { }
08        // Getter- und Setter-Methoden
09        public int getCategoryID() {
10              return categoryID;
11        }
12        public void setCategoryID(int categoryID) {
13              this.categoryID = categoryID;
14        }
15        public int getCategoryChildID() {
16              return categoryChildID;
17        }
18        public void setCategoryChildID(int categoryChildID) {
19              this.categoryChildID = categoryChildID;
20        }
21        public String getCategoryName() {
22              return categoryName;
23        }
24        public void setCategoryName(String categoryName) {
25              this.categoryName = categoryName;
26        }
27  }
```

**Listing 14: Beispiel einer Model-Klasse**

Aus Gründen der Flexibilität und der Erweiterbarkeit der Model-Klassen wurde sich dafür entschieden, die Product- und Catalog-Model-Klasse als Interface zu definieren. Die jeweils spezielle Implementierung des Model-Interfaces ist ähnlich der zuvor beschrieben Bean-Model-Klasse. Dies wird ebenfalls am Beispiel der folgenden Catalog-Model-Klasse einmal verdeutlicht.

```
01  import java.io.Serializable;
02  public interface ICategoryModel extends Serializable {
03        public void setCategoryID(int categoryID);
04        public int getCategoryID();
05        public void setCategoryName(String categoryName);
06        public String getCategoryName();
07        public void setCategoryChildID(int categoryChildID);
08        public int getCategoryChildID();
```

**Listing 15: Beispiel einer Model-Interface-Klasse**

Der Aufbau des Bundle gestaltet sich daher etwas anders als die der vorherigen Bundle. Es gibt ein Bundle für den Bundle-Activator, der den Product-Service und Catalog-Service registriert. Zudem gibt es jeweils ein Bundle für Product und Catalog. Die beiden Bundle sind ähnlich aufgebaut, sie beinhalten jeweils eine Model-Interface-Klasse, eine konkrete Implementierung des Model-Interfaces, ein Service-Interface und die konkrete Implementierung des Service-Interface. Die Methoden der zwei Service-Interfaces für Product und Catalog orientieren sich am CRUD-Prinzip. CRUD steht für create, read, update und delete. Übertragen auf das Product-Service-Interface heißt das, es wurden Methoden definiert, die das Erstellen, Lesen, Erneuern und Löschen eines, oder mehrere Produkte ermöglichen. Als Übergabe-Parameter oder als Rückgabewert dienen die Model-Klassen.

### Jetty-Bundle

Das Jetty-Bundle besitzt eine Activator-Klasse, welche den Jetty-Server bei Bundle-Start hochfährt, konkret wird eine statische Methode start() aufgerufen, welche sich in der eigentlichen Jetty-Server-Klasse befindet. Die Konfiguration des Jetty-Servers erfolgt programmatisch. Außerdem registriert die Activator-Klasse einen Service, welcher die Registrierung eines Servlets und einer WebApp am Jetty-Server selbst zulässt. Dies ist wichtig, um das Frontend am Jetty-Server anmelden zu können. Nachfolgend wird die start()-Methode der Jetty-Server-Klasse in Listing 16 dargestellt.

```
01  protected static void start(String serverHostAdress, int serverPort)
02              throws Exception {
03          // aktiviere Jetty-Log
04          Log.getLog().setDebugEnabled(true);
05          // erstelle neues Server-Objekt
06          server = new Server();
07          // erstelle Server-Connector
08          connector = new SelectChannelConnector();
09          connector.setPort(serverPort);
10          connector.setHost(serverHostAdress);
11          // erstelle Server-Handler-Collection
12          handlerCollection = new HandlerCollection();
13          contextHandlerCollection = new ContextHandlerCollection();
14          handlerCollection.setHandlers(new Handler[] {
15                  contextHandlerCollection} );
16          // fuege Connector und Handler zum Server hinzu
17          server.setConnectors(new Connector[] { connector });
18          server.setHandler(handlerCollection);
19          server.setStopAtShutdown(true);
20          server.setThreadPool(new QueuedThreadPool(20));
21          // starte Server
22          server.start();
23  }
```

**Listing 16: Start des Jetty-Server**

### Frontend-Bundle

In Punkt 4.2 Auswahl von Technologien wurde begründet, warum GWT als Technologie für die Frontendentwicklung ausgewählt wurde. Aufgrund des Umfangs hat sich der Verfasser der Bachelorarbeit dazu entschieden nur in groben Zügen, die Funktionalität von GWT und dem Frontend-Bundle zu beschreiben.

Zuerst wurden die GWT-Bibliotheken dem Frontend-Bundle hinzugefügt und durch einen Eintrag in die Manifest Datei des Bundle dem Klassenpfad hinzugefügt. Der so in Java geschriebene Quellcode wurde mit Hilfe eines Ant-Build-Skript zu JavaScript kompiliert. Durch die Einbindung von JavaScript ist es möglich, dynamisch Inhalte zu wechseln, ohne die Seite erneut laden zu müssen.

Als nächstes wurde eine HTML-Datei erstellt, welche zum Einen auf den kompilierten JavaScript-Code zugreift und zum Anderen für die grafische Formatierung eine CSS-Datei benutzt. Sowohl die HTML-Datei als auch der kompilierte JavaScript-Code greifen für das Layout beziehungsweise die Formatierung der grafischen Komponenten auf die CSS-Datei zurück. Um dies zu bewerkstelligen, wurden in der HTML-Seite sogenannte Div-Container beziehungsweise Div-Tags für den Aufbau der Seite verwandt. Dies hat den Vorteil, dass sich mit Hilfe der CSS-Datei das Layout der Seite ändern lässt. Nachfolgend wird die HTML-Seite im Quelltext dargestellt, siehe Listing 17 Beispiel einer Manifest-Datei.

```
01 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
02 <html>
03 <head>
04 <meta http-equiv="content-type" content="text/html; charset=UTF-8">
05 <meta name="gwt:property" content="locale=de_DE">
06 <title>Shop</title>
07 <!-- style sheet -->
08 <link type="text/css" rel="stylesheet" href="IgnitiSimpleShop.css">
09 <!-- GWT generated java script -->
10 <script type="text/javascript" language="javascript"
11       src="ignitisimpleshop/ignitisimpleshop.nocache.js"></script>
12 </head>
13 <body>
14       <center>
15             <!-- Layout  -->
16             <div class="global" id="global">
17                   <div class="cart" id="cart"></div>
18                   <div class="header" id="header"></div>
19                   <div class="center" id="center">
20                         <div class="nav" id="nav"></div>
21                         <div class="main" id="main"></div>
22                   </div>
23                   <div class="footer" id="footer"></div>
24             </div>
25       </center>
26       <!-- GWT history support -->
27       <iframe src="javascript:''" id="__gwt_historyFrame" tabIndex='-1'
28             style="position: absolute; width: 0; height: 0; border: 0">
29       </iframe>
30 </body>
31 </html>
```

**Listing 17: Frontend HTML-Seite**

Da die Webapplikation, also das Frontend, eine Mischung aus statischen und dynamischen Webinhalten ist, muss sie als WebAppContext beim Jetty-Server registriert werden. Dies wird realisiert durch den vom Jetty-Bundle angebotenen Service. In den meisten Fällen wird dazu eine WAR-Datei verwendet. Eine WAR-Datei ist eine im ZIP-Format gepackte Datei, mit der Dateiendung .war. Sie wird in den meisten Fällen dazu benutzt, um entwickelte Webanwendungen auf einen Webserver zu installieren. Zusätzlich ist es möglich, eine Webapplikation als WebAppContext am Jetty-Server zu registrieren, die keine gepackte WAR-Datei ist. Aus dem zeitlichen Vorteil heraus, hat sich der Verfasser dieser Bachelorarbeit gegen eine gepackte Variante der WAR-Datei entschieden.

Bei der Entwicklung des Frontend-Bundle stellte sich die Frage, ob es sinnvoll ist, die Produktbilder in der Datenbank zu speichern, das heißt sie für die Anzeige im Front-End aus der Datenbank zu lesen und ein Produktbild somit über die Product-Model-Klasse mitzuliefern. Aus Handhabungs- und Performance-Gründen stellte die Speicherung der Produktbilder in der Datenbank keine zufriedenstellende Lösung dar. Stattdessen wurde ein

Image-Bundle erstellt, welches sich über den Service des Jetty-Bundle als Servlet beim Jetty-Server registiert. Die Bilder liegen also auf den File-System des Hardware-Servers. Damit kann die Webapplikation des Front-End-Bundle mit Hilfe des Image-Servlets per GET-Request die Produktbilder beziehen. Somit werden in der Datenbank die Produktbilder nicht selbst gespeichert, sondern nur die Dateinamen der Produktbilder. Aus diesem Grund wurde eine Notfallfunktion in der Webapplikation integriert, falls die Webapplikation über das Image-Servlet kein Produktbild beziehen kann. Die Notfallfunktion sieht so aus, dass Anstelle des fehlenden Produktbildes ein so genanntes Dummy-Bild angezeigt wird, welches standardmäßig in die Webapplikation, beziehungsweise in das Frontend integriert wurde.
Außerdem war es notwendig, für den clientseitigen Code eigene Model-Klassen zu schreiben und zusätzlich Java-Klassen für die Umwandlung zu entwickeln. Der Grund hierfür war, dass beim Kompilieren des Java-Quellcodes zu JavaScript die clientseitigen Klassen angegeben werden müssen, die dann letztendlich umgewandelt werden. Zudem kann die Angabe der clientseitigen Klassen nur Java-Package-Weise gemacht werden. Da aber die Model-Klassen für Product, Catalog und Order nicht in den gleichen clientseitigen Java-Packages liegen und sich außerdem in anderen Bundle befinden, kann der GWT-Compiler, der über ein ANT-Build-Skript ausgeführt wird, nicht auf diese Java-Klassen zugreifen.
Zur Veranschaulichung wurden Bilder vom Frontend erstellt, diese befinden sich im Anhang Seite XVIII ff.

**Email- und Order-Bundle**

Nachfolgend wurden die beiden Bundle Email und Order erstellt. An der Stelle wird kurz beschrieben welchen Service die zwei Bundle anbieten.
Das Email-Bundle wurde so implementiert, dass es, ähnlich wie beim Database-Bundle, eine Verbindung für die Email liefert, beziehungsweise wird durch Methoden die Möglichkeit gegeben, Emails über die Protokolle POP3, IMAP und SMTP zu empfangen oder zu versenden.
Das Order-Bundle wurde so angedacht, dass es einen Service liefert, der eine Bestellung speichern kann, schon in der Datenbank gespeicherte Bestellungen auslesen kann, Bestellungen ändern kann und Bestellungen auch löschen kann. Allerdings wurde das Order-Bundle nach vorheriger Abstimmung mit dem Betreuer dieser Bachelorarbeit nicht implementiert.

### Bundle-Übersicht

Nach der Implementierung und einer kleinen Umstrukturierung der Bundles ergaben sich folgende Bundle-Abhängigkeiten, welche die nachfolgende Abbildung veranschaulicht.

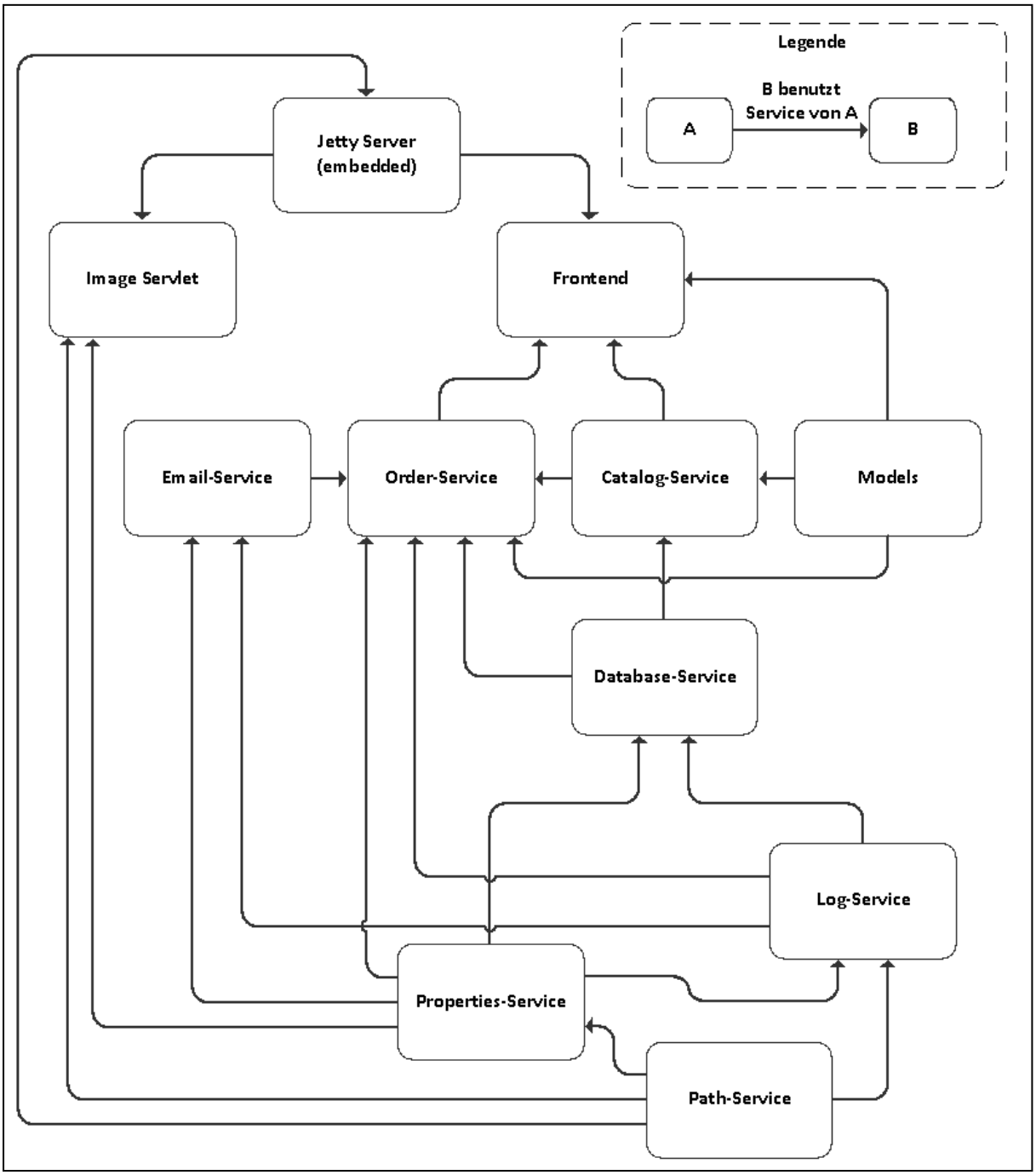



**Abbildung 13: Bundle-Abhängigkeiten**

**Core-Bundle**

Zusätzlich wurden für alle bisher entwickelten Services Notfallvarianten implementiert. Diese sollen dann greifen, wenn ein Bundle einen Service nutzten möchte, aber das Bundle, das den Service anbietet, nicht vorhanden ist.
Da mehrere OSGi-Services unter den gleichen Namen registriert werden können, wurde ein Service-Manager implementiert. Das Core-Bundle registriert den Service-Manager als OSGi-Service. Der Service-Manager enthält eine Methode chooseService(), welcher die Übergabe der gesuchten Service-Klasse und des Bundle-Context erfordert. Diese Methode funktioniert folgendermaßen: Zuerst wird per OSGi-ServiceTracker nach allen, bisher registrierten OSGi-Services gesucht. Der Name des Services wird über das übergebene Java-Klassen-Objekt ermittelt. Werden nun mehrere OSGi-Services gefunden, welche unter dem gleichen Namen registriert sind, wird per Auswahl-Mechanismus ein Service ausgewählt und zurückgegeben. Würde man per OSGi-ServiceTracker, mit der Methode getServices(), nicht nach allen verfügbaren OSGi-Services suchen, die unter demselben Namen registrierten sind, sondern mit der Methode getService() suchen, würde OSGi per eigenen Auswahl-Mechanismus einen Service zurückliefern. Wurde kein OSGi-Service gefunden, wird fünf Sekunden gewartet, ob sich während dieser Zeit ein Service, unter dem gesuchten Service-Namen, registriert. Ist dies auch nicht der Fall, wird auf die selbst implementierten Notfallvarianten zurückgegriffen.
Sowohl der Service-Manager, als auch die Notfallvarianten liegen im Core-Bundle. Desweiteren wurden alle bisherigen Service-Interfaces aus ihren Bundle entfernt und in das Core-Bundle transferiert. Auch alle Model-Klassen wurden in das Core-Bundle verlegt. Zudem war es notwendig alle Java-Packages des Core-Bundle in dem die Service-Interfaces und Model-Klassen liegen, über Einträge in der Manifest-Datei, zu exportieren. So erhalten die anderen Bundle Zugriff auf diese Klassen.

Außerdem wurde das Database-Bundle, mit einem Connection-Pool ausgestattet. Dieser Connection-Pool hält vorrätig eine definierte Anzahl von Verbindungen zum SQL-Server. Dadurch ist es dem Database-Service möglich, eine freie Verbindung aus dem Connection-Pool zu liefern. Weiterer Vorteil ist, dass die Verbindung zum SQL-Server nicht erst aufgebaut werden muss, da das Liefern einer gültigen und schon hergestellten Verbindung der Connection-Pool steuert. Für den Connection-Pool wurde auf die beiden Apache Commons Java-Bibliotheken Pool und DBCP zurückgegriffen.

# 5 Testen

In diesem Kapitel wurde sich darauf beschränkt für den entwickelten Prototyp geeignete Test-Arten aufzuführen und kurz zu beschreiben. Da im vorgegeben Bearbeitungszeitraum dieser Bachelorarbeit aus zeitlichen Gründen keine Tests durchgeführt werden konnten.

Zum Einen könnten sogenannte Modul-Tests erstellt und durchgeführt werden. Bei Modul-Tests werden abgrenzbare Teile des Gesamtsystems auf Funktionalität hin getestet, zum Beispiel: Teilprogramme, Module, einzelne Systemkomponenten, oder einzelne Klassen. Ziel ist es diese abgrenzbaren Teile isoliert von anderen Teilen zu testen. Übertragen auf den entwickelten Prototyp würde sich daher anbieten die Modul-Tests auf Bundle-Ebene durchzuführen, also die einzelnen OSGi-Bundle, auf ihre Funktionalität hin zu testen. Dazu wird aber ein geeignetes Test-Framework benötigt. Da für die Entwicklung des Prototyps, die Programmsprache Java verwendet wurde, würde sich daher die Verwendung des JUnit-Frameworks anbieten, welches in der Praxis oft für das Testen von Java-Programmen verwendet wird. Zudem gibt es verschiedene OSGi-spezifische Test-Frameworks. Zum Beispiel besitzt die OSGi-Implementierung von Spring DM eine eigene Test Suite namens OSGi-Mocks. Zusätzlich bieten Test-Frameworks wie „luminis OSGi testing framework“ und „DA-Testing Framework“ eine Kompatibilität für Tests an, welche von der OSGi-Implementierung unabhängig ist.
Desweiteren könnten Leistungs-Tests durchgeführt werden, zum Beispiel am Datenbank-Server und am verwendeten Webserver. Hier könnte eine weitere Aufspaltung des Leistungs-Tests in einen Stress-Tests und einen Last-Tests erfolgen. Zudem könnte ein Funktions-Test des Front-Ends des Prototyps erfolgen, also zum Beispiel: Ob alle Buttons wie erwartet funktionieren, ob alle Elemente angezeigt werden, wie Preise, Bilder, Tabellen, Links usw. Für das Testen des Front-Ends würde sich daher die Selenium Testsuite anbietet, welche auf das auch automatisierte Testen von Webapplikation spezialisiert ist. Zudem bietet die Selenium Testsuite Unterstützung für verschiedenste Internetbrowser, Betriebssysteme, Programmiersprachen und andere Test-Frameworks. Zusätzlich wären für den entwickelten Prototyp Integrations-Tests sinnvoll. Mit diesen Tests sollen voneinander abhängige Komponenten getestet werden. Auf den Prototypen übertragen könnte zum Beispiel getestet werden, ob die angezeigten Produktdaten im Front-End übereinstimmen mit den gespeicherten Datensätzen in der Datenbank.

# 6 Zusammenfassung und Ausblick

Nachfolgend wird eine Zusammenfassung der erreichten Ergebnisse und darauf folgend ein Ausblick gegeben.

## 6.1 Zusammenfassung

Ziel dieser Bachelorarbeit war es, einen Prototyp eines Shop-Systems auf der Basis von OSGi-Komponenten unter dem Gesichtspunkt eines modularen Softwaresystems zu entwickeln.
Dazu wurde im theoretischen Teil dieser Bachelorarbeit aufgezeigt, was ein Shop-System und ein modulares Softwaresystem charakterisiert und wie OSGi in seinen Grundzügen funktioniert und aufgebaut ist. Zudem wurde in Punkt 0.4 eine thematische Abgrenzung vorgenommen. Desweiteren wurde ein Variantenvergleich von bestehende Shop-Systemen vorgenommen, um diese auf die Eignung eines modularen Softwaresystems hin zu überprüfen. Durch den Vergleich wurde festgestellt, dass keines der untersuchten Shop-Systeme die Anforderungen an ein vollständig modulares Shop-System erfüllt. Darauf folgend wurde in Kapitel 3 die mögliche Eignung von OSGi erörtert.
Durch die Entwicklung eines Prototyps wurde untersucht, ob OSGi die Eignung besitzt, also die Anforderungen die an ein modulares Softwaresystem gestellt werden, erfüllt. Nachfolgend werden die erzielten Ergebnisse beschrieben. Allerdings muss dabei erwähnt werden, dass eine Fertigstellung des Prototyps innerhalb des gegebenen Bearbeitungszeitraumes nicht erzielt werden konnte. Dennoch wurden die eigentlichen Teilziele bei der Implementierung, erreicht.
Der Verfasser dieser Bachelorarbeit ist zu dem Schluss gekommen, dass sich OSGi, für die Entwicklung eines modularen Softwaresystems, prinzipiell eignet. Da OSGi viele Mechanismen, wie zum Beispiel das Bundle-Konzept, OSGi-Services, OSGi-Registry, ServiceTracker und deklarative Services, spezifiziert hat, die die Anforderungen an ein modulares Softwaresystem erfüllen. Allerdings hat sich besonders bei der Implementierung, gezeigt, dass sich mit zunehmender Modul-Zahl beziehungsweise Bundle-Anzahl die Komplexität der Abhängigkeiten drastisch verstärkt. Trotzdessen konnte durch die Entwicklung des Prototyps bewiesen werden, dass sich die einzelnen Bestandteile im Fall eines Shop-Systems in ein OSGi-Bundle verpacken lassen. Allerdings mussten hierfür einige technische Hindernisse überwunden werden. Das Database-Bundle stellt hierbei eine Ausnahme dar, es wäre zwar möglich gewesen eine eingebettete Datenbank, wie zum Beispiel Apache Derby zu verwenden, allerdings wurde sich aus verschiedenen Gründen dagegen entschieden.

Weiteres Ziel war es das Pluggable-Konzept umzusetzen. Es wurde zu dem Schluss gekommen, dass OSGi eine gute Eignung dafür besitzt. Besonders durch OSGi-Bundle-Konzept bietet sich die Tauglichkeit für das Pluggable-Konzept an. Allerdings muss gesagt werden, dass bei der Programmierung darauf geachtet werden muss, die Bundles robust zu implementieren und außerdem die Implementierung von eigenen Mechanismen, wie zum Beispiel eines Service-Managers, ratsam ist, um die fehlerfreie Funktion des Pluggable-Konzeptes zur Laufzeit zu ermöglichen.
Zusätzlich hat sich die Entwicklung des Front-Ends mit GWT als sehr zeitraubend herausgestellt. Aus Sicht des Autors dieser Bachelorarbeit wäre eine Modularisierung des Front-Ends mit GWT sehr schwierig und zeitaufwändig. Da mit OSGi nicht nur die vertikale Änderung der Schichten des Softwaresystem möglich ist, sondern auch eine horizontale Schichtenänderung bis in die untersten Schichten möglich ist, würde sich anbieten wenn das Frontend sich dynamisch aus mehreren Bestandteilen zusammensetzen würde. Damit ist GWT für die Front-End-Entwicklung nur für kleinere Projektumfänge geeignet.

## 6.2 Ausblick

Der bisher entwickelte Prototyp könnte weiterentwickelt werden. Besonders im Service-Auswahl-Mechanismus des Service-Managers steckt noch viel Verbesserungspotential. Zudem wäre es ratsam, sich für eine andere Technologie für das Front-End zu entscheiden. Allerdings stellt sich dabei die Frage, ob dies dann überhaupt möglich ist, die andere Variante des Front-Ends als ein Bundle oder auf mehrere Bundle verteilt zu realisieren. Zusätzlich könnte ein grafisches Back-End als OSGi-Bundle implementiert werden. Außerdem könnte das Datenbankmodell verbessert werden. Des Weiteren könnten noch entsprechende OSGi-Bundle-Tests geschrieben werden, zum Beispiel mit dem Test-Framework JUnit. Ebenfalls wäre die Verwendung des OSGi-Loggers möglich. Dieser ist besonders hilfreich, wenn man automatisch den Zustandswechsel eines Bundles protokollieren möchte. Generell können die Funktionalitäten des Prototyps, also die des Feature-Bundle, erhöht werden. Darüber hinaus wäre es denkbar, den Prototypen als verteiltes System zu entwickeln, um eine Skalierbarkeit des Gesamtsystems zu erreichen. Zusätzlich könnte der Prototyp an ein Content-Management und ein Warenwirtschaftssystem angeschlossen werden, beziehungsweise könnten dafür Anbindungen implementiert werden. Außerdem wäre es denkbar, aufbauend auf den gewonnen Erkenntnissen aus der Prototypentwicklung, erst einmal ein Shop-System im kleinen Rahmen zu entwickeln.

# Glossar

| | |
|---|---|
| **2-Level-Kategorie** | Dieser selbstdefinierte Begriff steht für ein Kategorie-System, welches aus mehreren Primärkategorien bestehen kann, die wiederum mehrere Unterkategorien beinhalten können. |
| **Check-out** (e-Commerce) | Ist das Abschließen eines elektronischen Kaufvorgangs im Internet. |
| **Cross-Browser-Kompatibilität** | Darunter versteht man die gleiche Ausgabe einer Webseite in verschiedenen Internetbrowsern. |
| **Cross-selling** (Onlineshop) | Ist das Anzeigen von ähnlichen Produkten, Waren oder Dienstleistungen, zum eigentlich angeboten, bzw. angezeigten Hauptprodukt, mit dem Ziel der Umsatzsteigerung pro Kunde. |
| **Datenbankcluster** | Ist eine Sammlung von Datenbanken, auf welche über einen einzigen laufenden Server zugegriffen werden kann. |
| **Datenbanksystem** | Beschreibt ein Softwaresystem, welches die Verwaltung der Daten in einer Datenbank übernimmt. |
| **e-Commerce** | Ein sehr weitläufiger Begriff in der Literatur. Prinzipiell kann man sagen, es ist der elektronische Handel zwischen 2 Unternehmen oder zwischen einem Unternehmen und dem Kunden mit all seinen Begleiterscheinungen, wie zum Beispiel Kundenservice, Geschäftsbeziehungen, Kunden Informationsbeschaffung, Kundenberatung usw. |
| **E-Shop** | Ein elektronischer Laden der reelle Waren und digitale Produkte im Internet anbietet. |
| **Enterprise-Lösung** (Software) | Bezeichnet eine kostenpflichtige Version einer Software, die für Großkunden, bzw. Großunternehmen optimiert ist. |

| | |
|---|---|
| **Internationalisierung** (Softwareentwicklung) | Bedeutet, eine Software so zu entwickeln, dass diese einfach an eine andere Sprache und länderspezifische Eigenheiten angepasst werden kann. |
| **JavaScript** | JavaScript ist eine Skriptsprache. Häufiges Einsatzgebiet ist das erzeugen dynamischer Inhalte von Webseiten. |
| **JUnit** | Bezeichnet ein Framework, welches zum Testen von Programmen gedacht ist, die in der Programmiersprache Java geschrieben wurden. |
| **MySQL** | Ist ein relationales Datenbankverwaltungssystem. |
| **Onlineshop** | Siehe Begriff E-Shop. |
| **Premium-Lösung** (Software) | Bezeichnet meist eine kostenpflichte Version einer Software, welche zusätzliche Funktionalitäten bietet, zur Grundversion. |
| **Simple Article** | Dieser selbstdefinierte Begriff steht für die Anzeige eines Produktes in einem Onlineshops. Dieses Produkt besitzt lediglich eine minimale Beschreibung, die für ein Produkt zwingend erforderlich ist: Einen Produktnamen, einen Produktpreis und eine kurz Produktbeschreibung. |
| **Skriptsprachen** | Als Skriptsprachen werden Programmiersprachen bezeichnet, welche für relativ kleine und überschaubare Programmierprojekte gedacht sind. Programme die in Skriptsprachen geschrieben sind, werden als Skripte bezeichnet. |

# Literaturverzeichnis

**BROY, MANFRED/RUMPE, BERNHARD (2007):**
Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung; URL: <http://www.springerlink.com/content/y5u61n7352006g8u/>; verfügbar am 29.06.2011

**FUNKE, HOLGER (2009):**
Das OSGi Framework; URL: <http://it-republik.de/jaxenter/artikel/Das-OSGi-Framework-2221.html>; verfügbar am 30.06.2011

**Google (2011):**
Google Web Toolkit; URL: <http://code.google.com/webtoolkit/>; verfügbar am 16.07.2011

**HORN, TORSTEN (2009):**
OSGi: Dynamisches Komponentensystem für Java; URL: <http://www.torsten-horn.de/techdocs/java-osgi.htm>; verfügbar am 31.06.2011

**Internetagentur 4Success (2011):**
Onlineshop; URL: <http://www.internetseiten4business.de/internet-lexikon/onlineshop/>; verfügbar am 27.06.2011

**ITos GmbH (2011):**
Web-Applikation ; URL: <http://www.itos-gmbh.com/websolutions.html>; verfügbar am 15.07.2011

**KUPPER, RAMON (2011):**
Shop-Architektur; URL: <http://www.ecommerce-info.de/onlineshop/shop-architektur/>; verfügbar am 27.06.2011

**MARR, STEFAN (2002):**
Modulare Software Architektur und Abstrakte Datentypen; URL: <http://www.ingentaconnect.com/content/klu/287/2007/00000030/00000001/00000124>; verfügbar am 01.07.2011

**NEUMANN, ALEXANDER (2009):**
Open Service Gateway: 10 Jahre OSGi Alliance; URL: <http://www.heise.de/developer/meldung/Open-Service-Gateway-10-Jahre-OSGi-Alliance-196287.html>; verfügbar am 31.06.2011

**OSGi Alliance (2009):**
OSGi Service Platform - Core Specification - The OSGi Alliance - Release 4, Version 4.2; URL: < http://www.osgi.org/Download/File?url=/download/r4v42/r4.core.pdf>; verfügbar am 05.07.2011

**OSGi Alliance (2011a):**
OSGi Alliance Mission; URL: <http://www.osgi.org/About/Mission>; verfügbar am 03.07.2011

**OSGi Alliance (2011b):**
About the OSGi Alliance; URL: <http://www.osgi.org/About/HomePage>; verfügbar am 03.07.2011

**OSGi Alliance (2011c):**
Members; URL: <http://www.osgi.org/About/Members>; verfügbar am 03.07.2011

**SCHÄFER, TORSTEN (2011):**
Modulare Software; URL: <http://www.thorsten-schaefer.com/de/glossar/modulare-software.html>; verfügbar am 27.06.2011

**SCHWERD, UDO (2011):**
Architektur eines Online-Shop; URL: <http://www.iyotta.de/index.php/geld-verdienen-im-internet/online-shops/358-shop-architektur.html>; verfügbar am 29.06.2011

**SEEBERGER, HEIKO (2008):**
Erste Schritte mit OSGI Teil I und II; URL: <http://entwickler.de/zonen/portale/psecom,id,101,online,2078,.html>; verfügbar am 30.06.2011

**STEYER, RALPH (2007):**
Das Google Web Toolkit URL: <http://www.contentmanager.de/magazin/artikel_1292_google_web_toolkit_gwt_ajax_java.html>; verfügbar am 17.07.2011

**WEBAGENCY Business Unit der Consileon Business Consultancy GmbH (2011a):** Was ist e-Commerce? (Praxisorientierte Definition); URL: <http://www.webagency.de/infopool/e-commerce-knowhow/ak981030.htm>; verfügbar am 29.06.2011

**WEBAGENCY Business Unit der Consileon Business Consultancy GmbH (2011b):** Was ist e-Commerce? (Akademische Definition); URL: <http://www.webagency.de/infopool/e-commerce-knowhow/ak981021.htm>; verfügbar am 29.06.2011

# Anhang

## Anlage 1: Variationsvergleich

## Blatt 1

**Tabelle - Teil 1**

| Shop-System / Kriterium | 1&1 Perfect Shop | 1&1 Business Shop | 1&1 Business Pro Shop | Oxid Community Edition 4 | Oxid Professional Edition 4 | Oxid Enterprise Edition 4 |
|---|---|---|---|---|---|---|
| **Kundenzahl** | k.A. | k.A. | k.A. | k.A. | k.A. | k.A. |
| **Produktzahl** | auf 1.000 begrenzt | auf 10.000 begrenzt | auf 30.000 begrenzt | unbegrenzt | unbegrenzt | unbegrenzt |
| **Multisite** | nein | nein | nein | k.A. | k.A. | ja |
| **Mehrsprachigkeit** | nein | ja | ja | ja | ja | ja |
| **Erweiterbar** | nein | nein | nein | ja | Ja | ja |
| **Als verteiltes System** | nein | nein | nein | nein | nein | nein |
| **Skalierbar** | nein | nein | nein | nein | nein | ja, aber nur Datenbankcluster |
| **Bestellungen pro Stunde*** | k.A. | k.A. | k.A. | k.A. | k.A. | k.A. |
| **Technologien** | k.A. | k.A. | k.A. | PHP, MySQL | PHP, MySQL | PHP, MySQL |
| **Zusatz** | Einrichtungsassistent, Shop-Designer | Einrichtungsassistent, Shop-Designer | Einrichtungsassistent, Shop-Designer | kein WYSIWYG-Editor | Upgrade auf Enterprise Version möglich | je nach Lizenz 1-15 Prozessorkerne |
| **Preis** | 39,99 €/Monat | 39,99 €/Monat | 69,99 €/Monat | kostenlos | ca. 3000 € zzgl. Mwst. | einmalige Lizenz ab 15.000 €/Monat |

* Wert kann nach oben und unten abweichen – je nach verwendeter Hardware

## Anlage 1: Variationsvergleich

## Blatt 2

### Tabelle - Teil 2

| Shop-System / Kriterium | JamJam Internet-Shop | VirtueMart | Apache OFBiz | 123JavaShop | Intershop Enfinity Suite 6 |
|---|---|---|---|---|---|
| **Kundenzahl** | k.A. | k.A. | unbegrenzt | unbegrenzt | unbegrenzt |
| **Produktzahl** | k.A. | k.A | unbegrenzt | unbegrenzt | unbegrenzt |
| **Multisite** | nein | nein | k.A. | nein | ja |
| **Mehrsprachigkeit** | nein | ja | ja | k.A. | ja |
| **Erweiterbar** | k.A. | ja | ja | ja | ja |
| **Als verteiltes System** | nein | k.A. | ja | nein | ja |
| **Skalierbar** | nein | nein | k.A. | nein | ja |
| **Bestellungen pro Stunde*** | k.A. | k.A. | k.A. | k.A. | 50.000 |
| **Technologien** | k.A. | PHP, CSS, MySQL, Joomla | Java, XML, SQL | Java, JSP/JSTL, SQL (embedded) | k.A. |
| **Zusatz** | - | nutzt Joomla als CMS, basiert auf phpShop | - | - | - |
| **Preis** | auf Anfrage | kostenlos | kostenlos | kostenlos | k.A. |

* Wert kann nach oben und unten abweichen – je nach verwendeter Hardware

## Anlage 1: Variationsvergleich

## Blatt 3

**Tabelle - Teil 3**

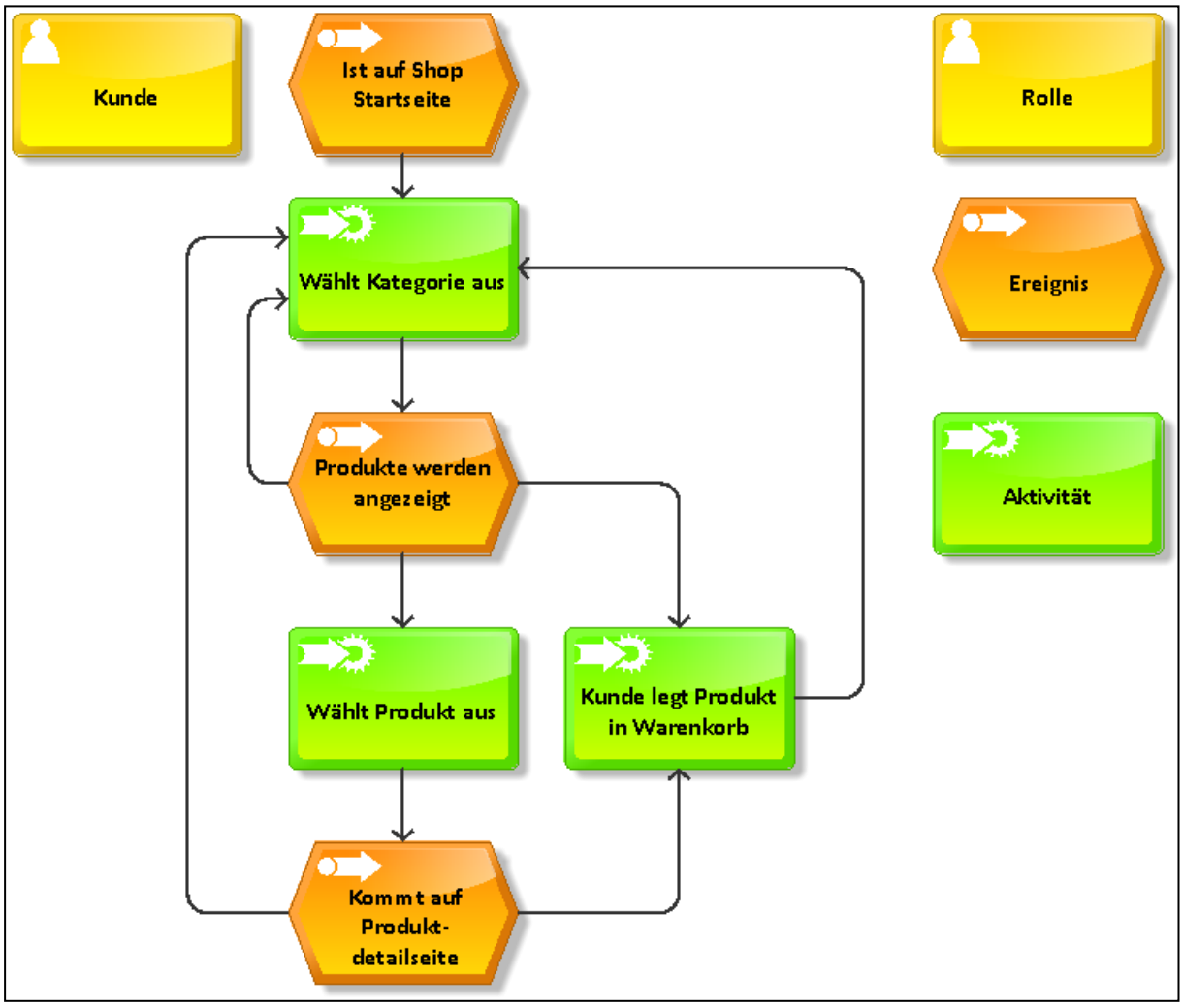
| Shop-System / Kriterium | Magento Community Edition | Magento Professional Edition | Magento Enteprise Edition | ePages Base | ePages Flex | ePages Enterpise S | ePages Enterprise M | ePages Enterprise L |
|---|---|---|---|---|---|---|---|---|
| **Kundenzahl** | unbegrenzt | unbegrenzt | unbegrenzt | unbegrenzt | unbegrenzt | unbegrenzt | unbegrenzt | unbegrenzt |
| **Produktzahl** | unbegrenzt | unbegrenzt | unbegrenzt | unbegrenzt | unbegrenzt | unbegrenzt | unbegrenzt | unbegrenzt |
| **Multisite** | k.A. | k.A. | ja | nein | nein | 1 pro Unternehmen | 1 pro Unternehmen | 2 pro Unternehmen |
| **Mehrsprachigkeit** | ja | ja | ja | ja | ja | 2 Sprachen | unbegrenzt | unbegrenzt |
| **Erweiterbar** | ja, | ja | ja | nein | ja | ja | ja | ja |
| **Als verteiltes System** | k.A. | k.A. | ja | k.A. | k.A. | ja | ja | ja |
| **Skalierbar** | ja | ja | ja | k.A. | k.A. | ja, Aufteilung auf mehrere Applikation-Server | ja, Aufteilung auf mehrere Applikation-Server | ja, Aufteilung auf mehrere Applikation-Server |
| **Bestellungen pro Stunde*** | k.A. | k.A. | ca. 80.000 | k.A. | k.A. | k.A. | k.A. | k.A. |
| **Technologien** | PHP (Zend Framework), MySQL | PHP (Zend Framework), MySQL | PHP (Zend Framework), MySQL | Perl, SQL | Perl, SQL | Perl, SQL | Perl, SQL | Perl, SQL |
| **Zusatz** | kein Support | Upgrade auf Enterprise-Version möglich | - | WYSIWYG-Editor, Content-Management-Editor, Sybase ASE (Datenbankserver) | Auswahl aus standardmäßigen Funktionen | Anzahl Applikation-Server 4 | Anzahl Applikation-Server 12 | Anzahl Applikation-Server mehr als 12 |
| **Preis** | kostenlos | ca. 2000 €/Monat | ca. 9000 €/Monat | 9,90 €/Monat | 80,00 €/Monat | 120,00 €/Monat | 240,00 €/Monat | 480,00 €/Monat |

* Wert kann nach oben und unten abweichen – je nach verwendeter Hardware

## Anlage 2: Front-End Anwendungsfalldiagramme

### Blatt 1

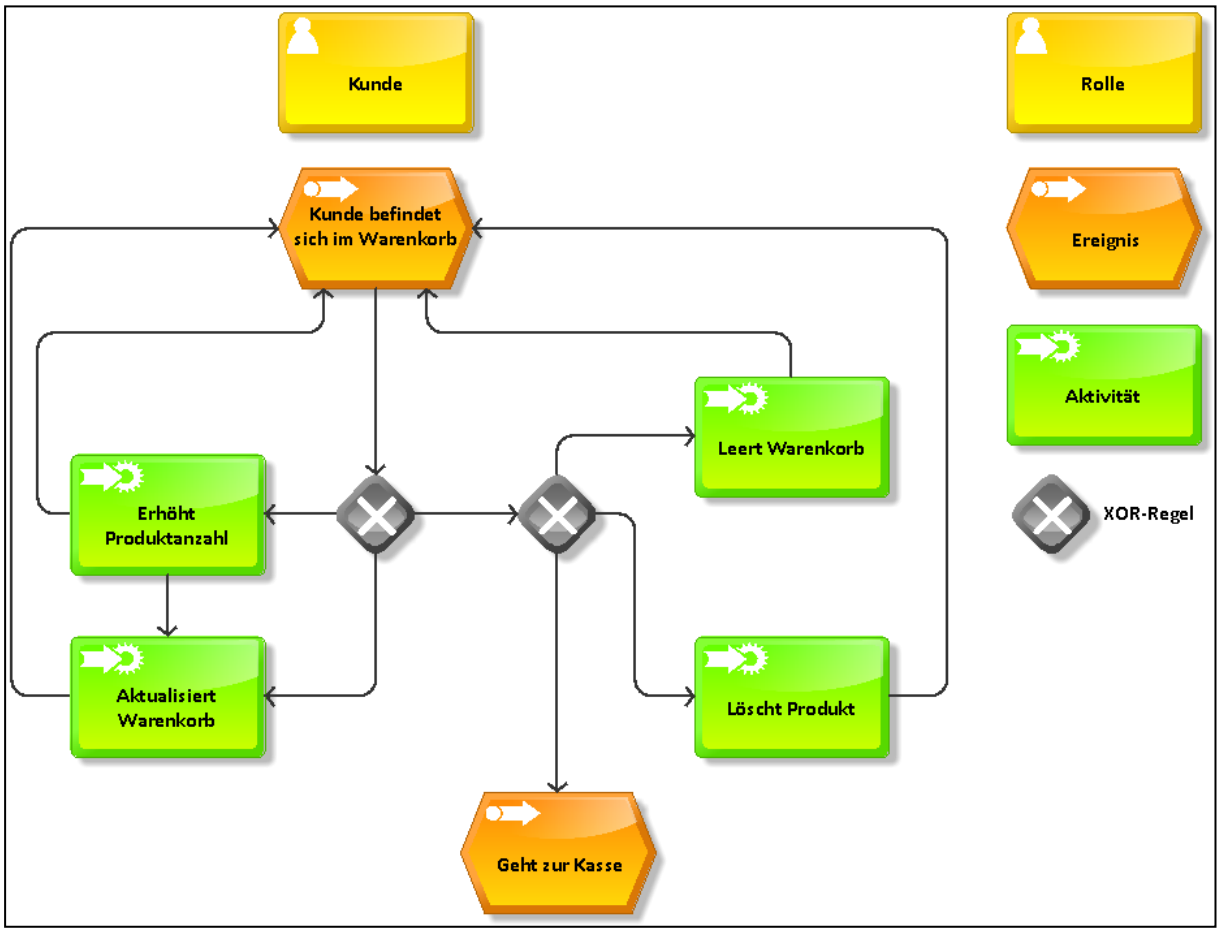
**Auswahl**



## Anlage 2: Front-End Anwendungsfalldiagramme

### Blatt 2

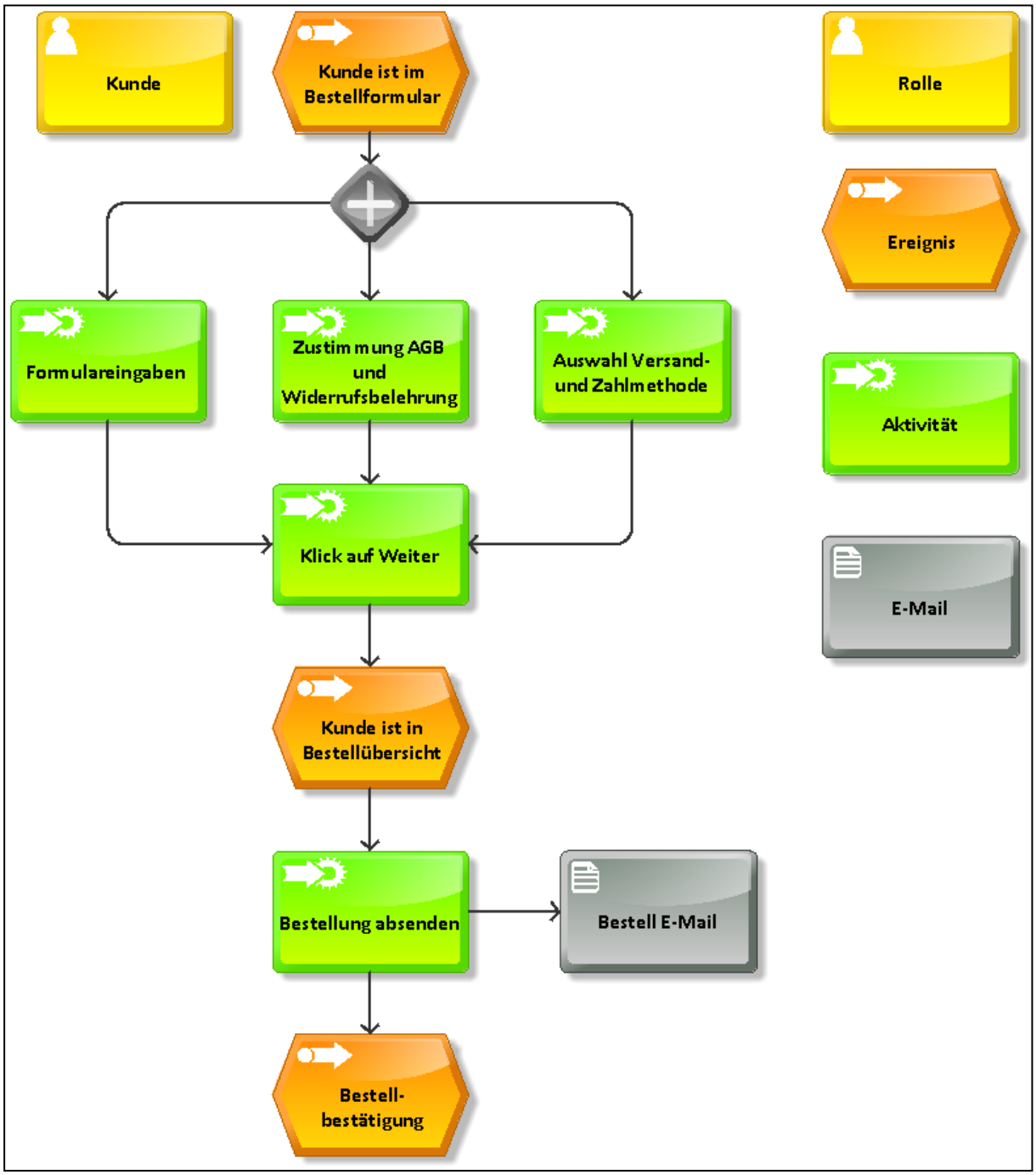
**Warenkorb**



## Anlage 2: Front-End Anwendungsfalldiagramme

### Blatt 3

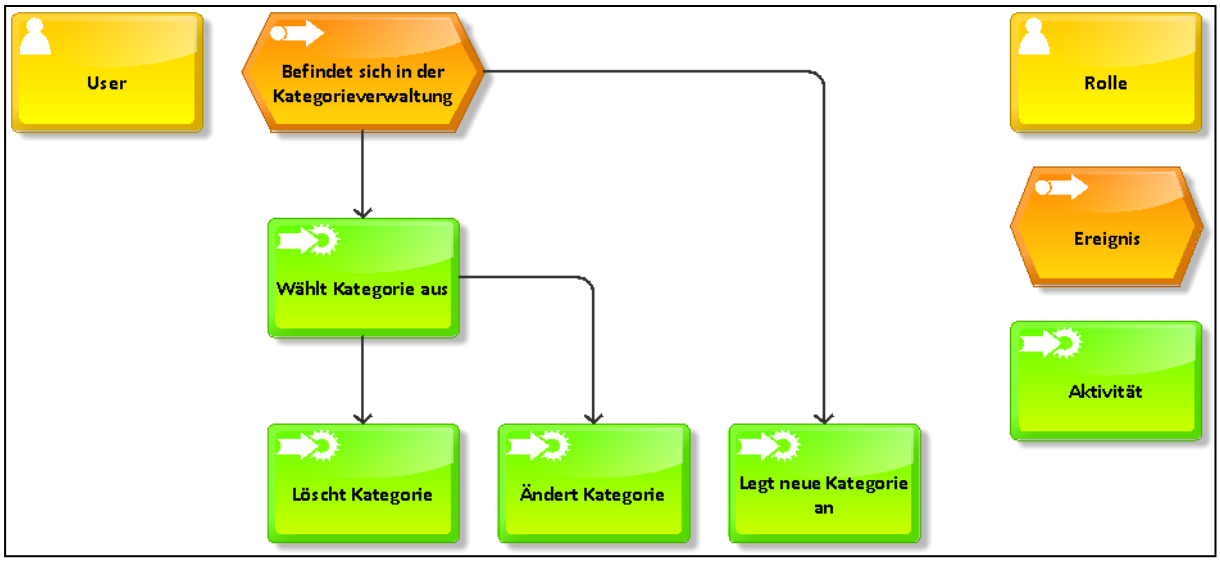
**Check-out**



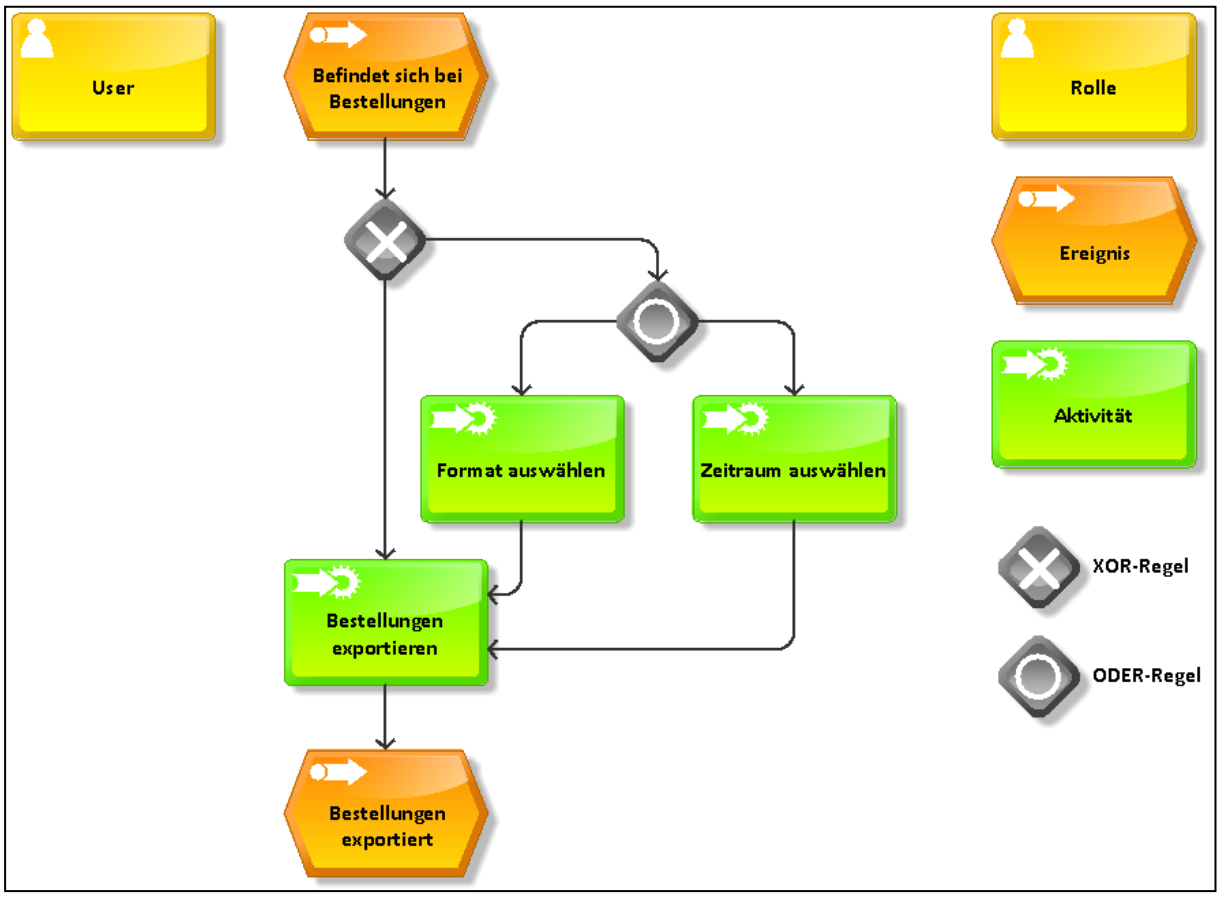
## Anlage 3: Back-End Anwendungsfalldiagramme

### Blatt 1

**Kategorieverwaltung**



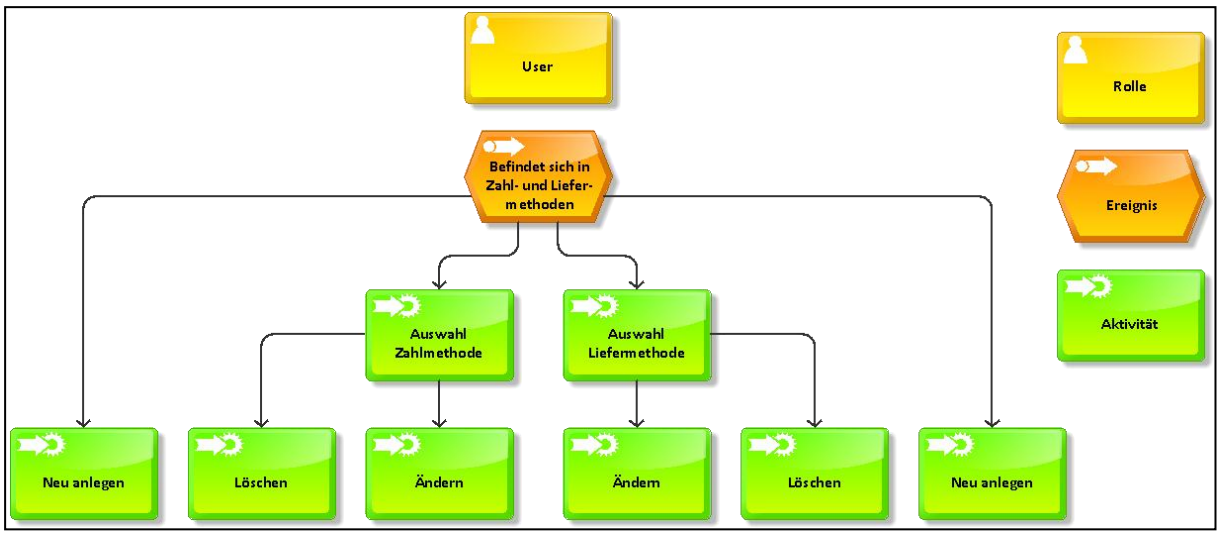

**Bestellung**



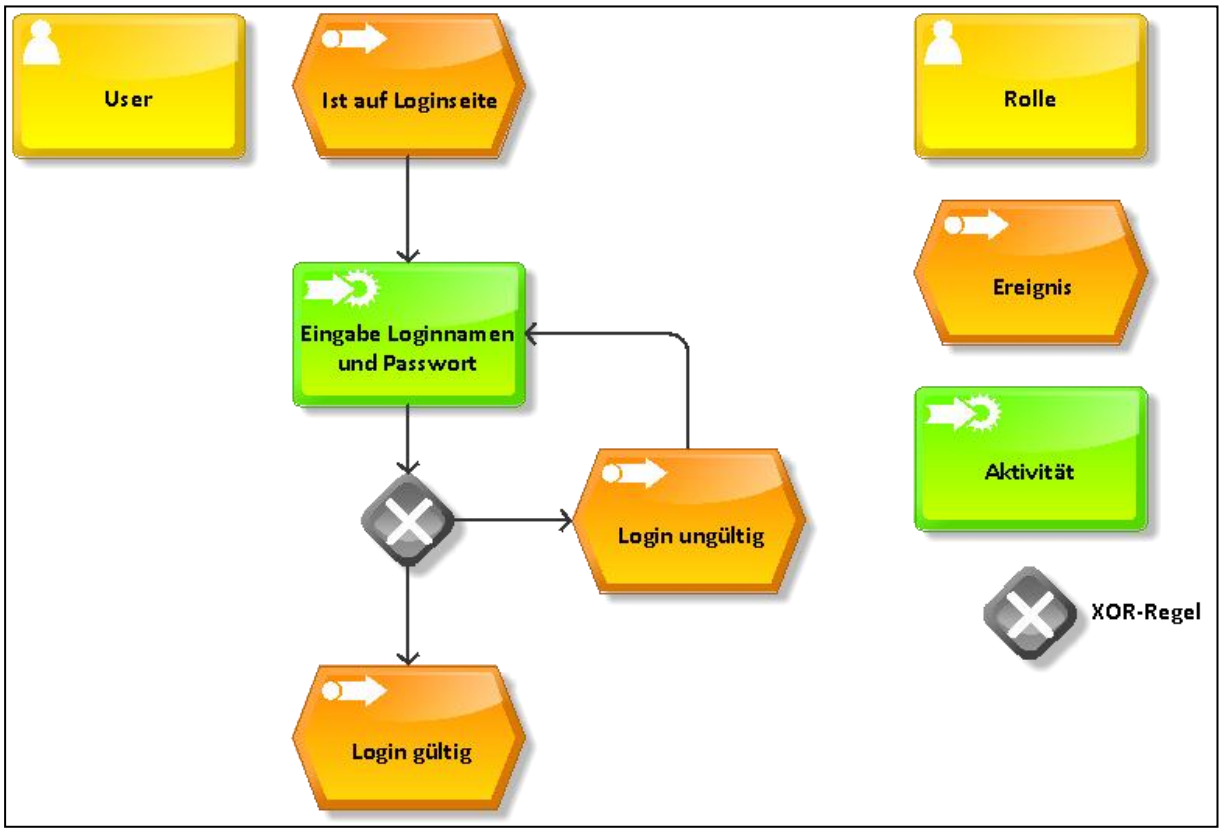
## Anlage 3: Back-End Anwendungsfalldiagramme

### Blatt 2

**Versand- und Liefermethoden**



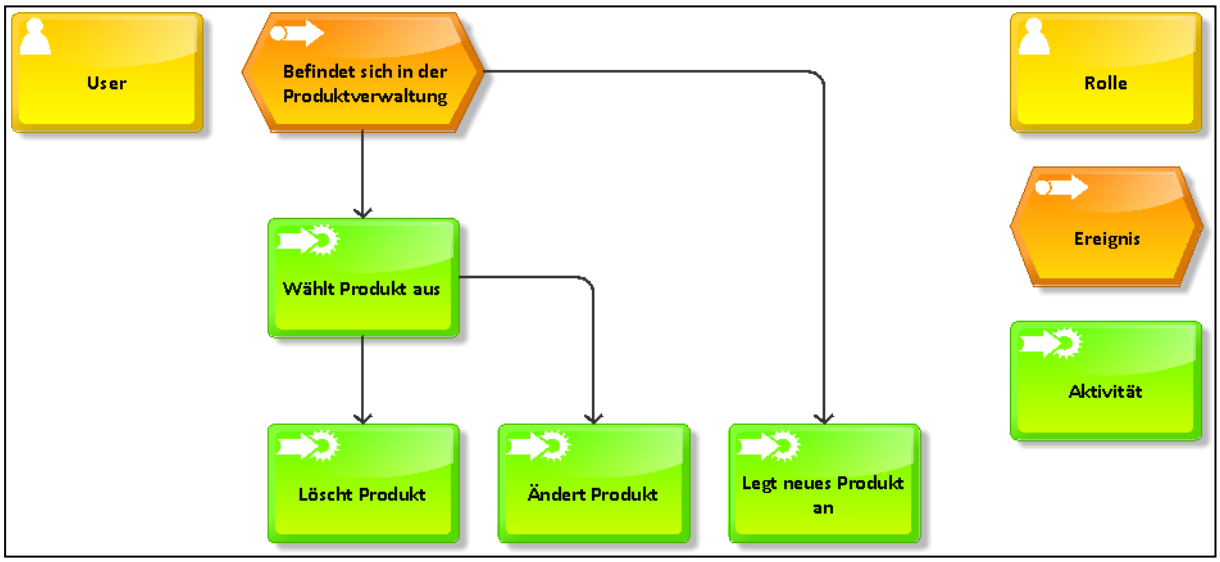

**Login**



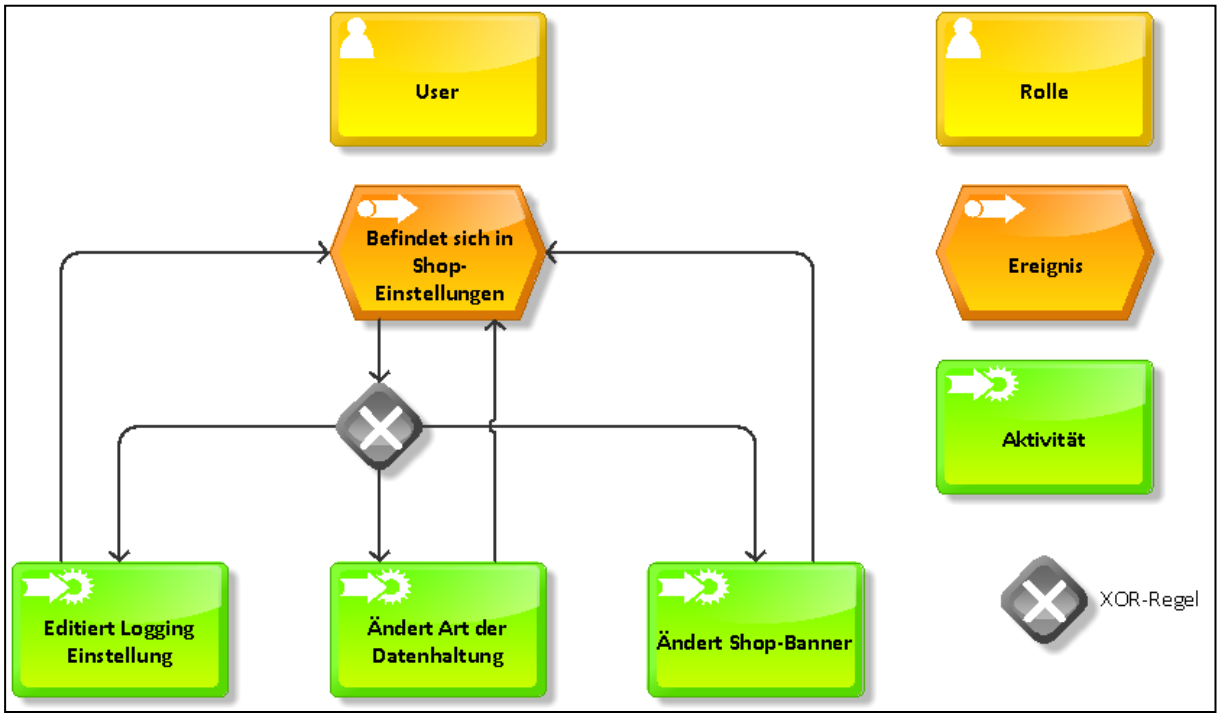
## Anlage 3: Back-End Anwendungsfalldiagramme

### Blatt 3

**Produktverwaltung**



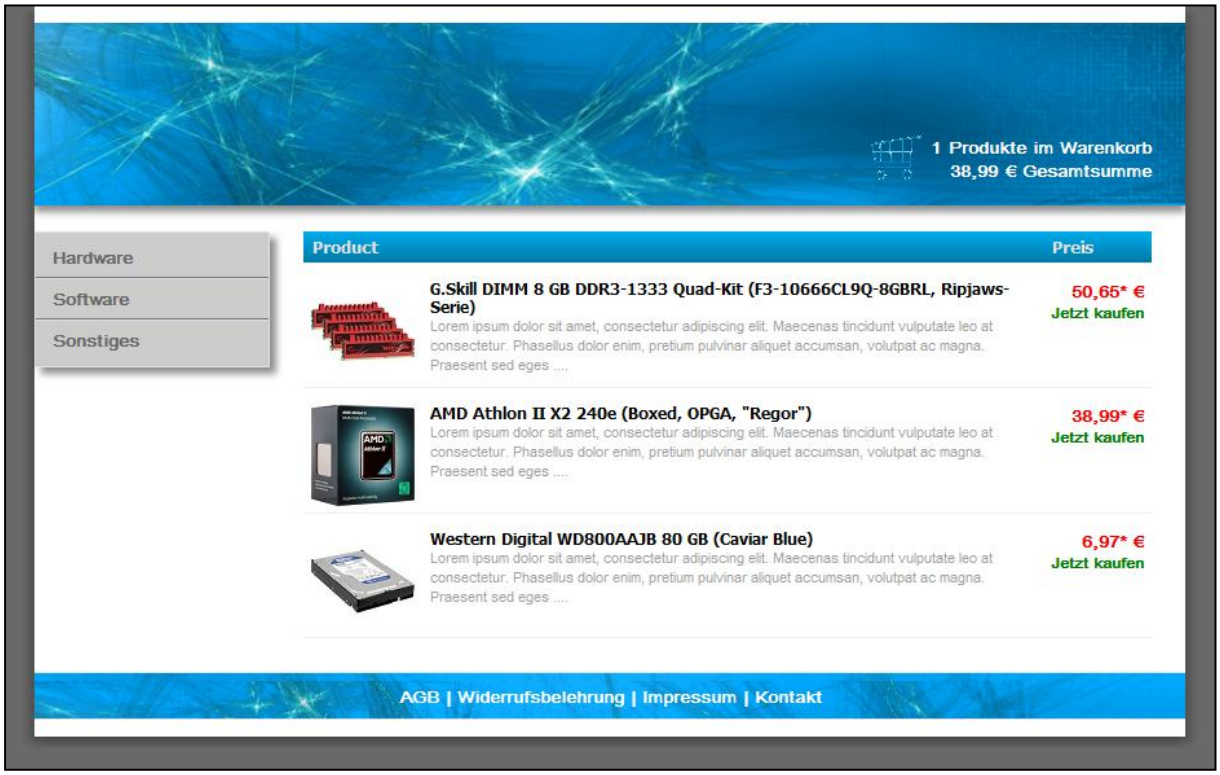

**Shop-Einstellungen**



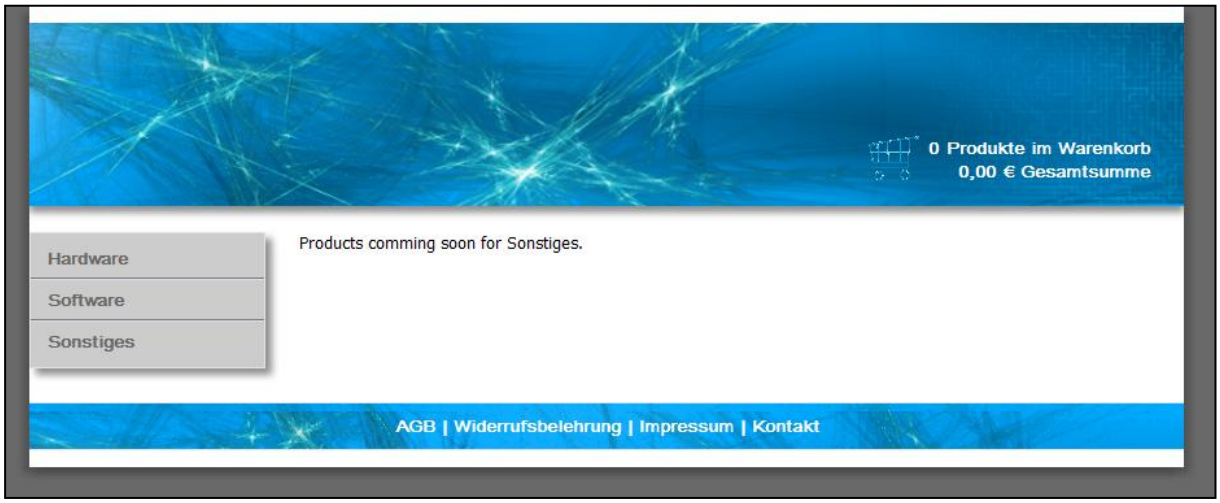
# Anlage 4: Front-End-Bilder

## Blatt 1

### Produktübersicht 1



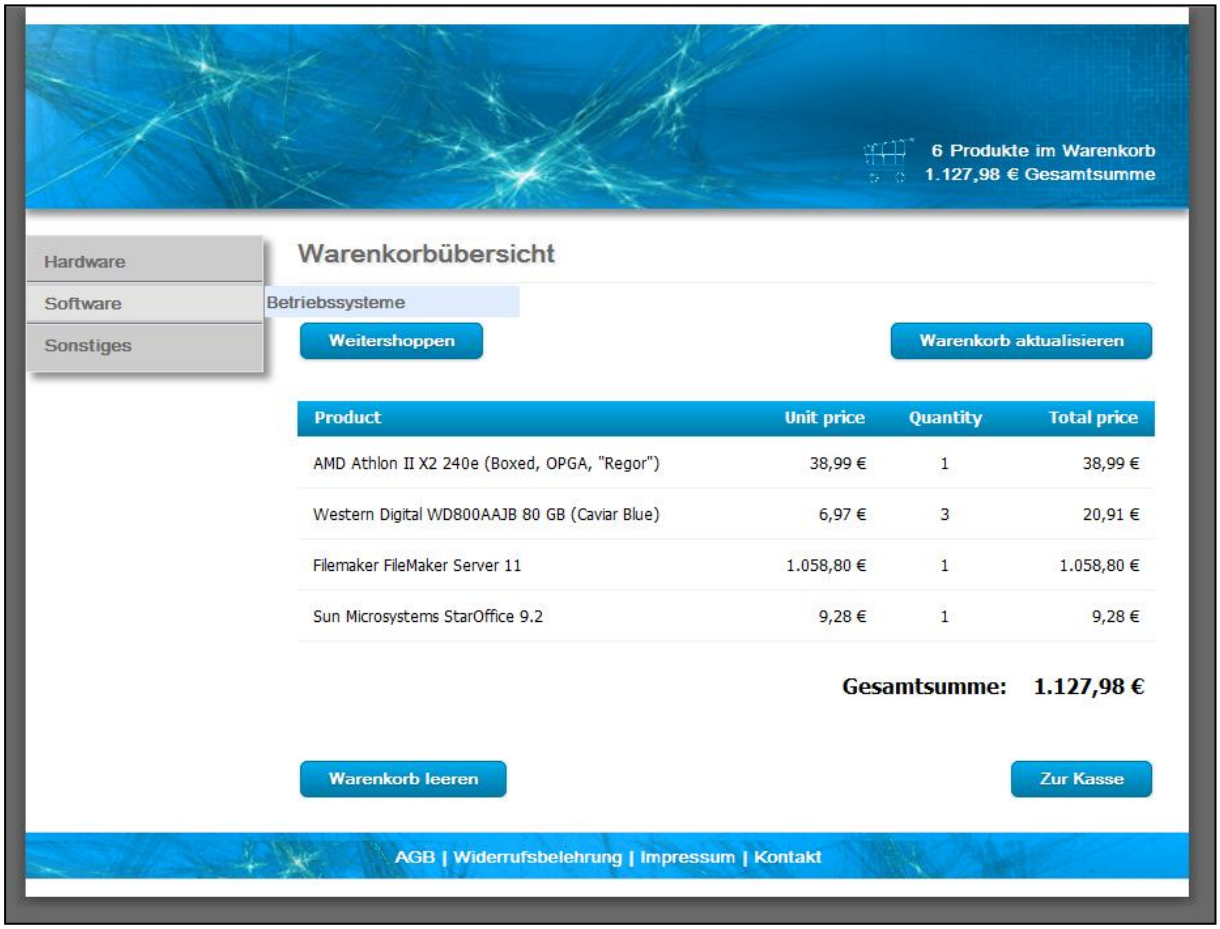

### Produktübersicht 2



## Anlage 4: Front-End-Bilder

### Blatt 2

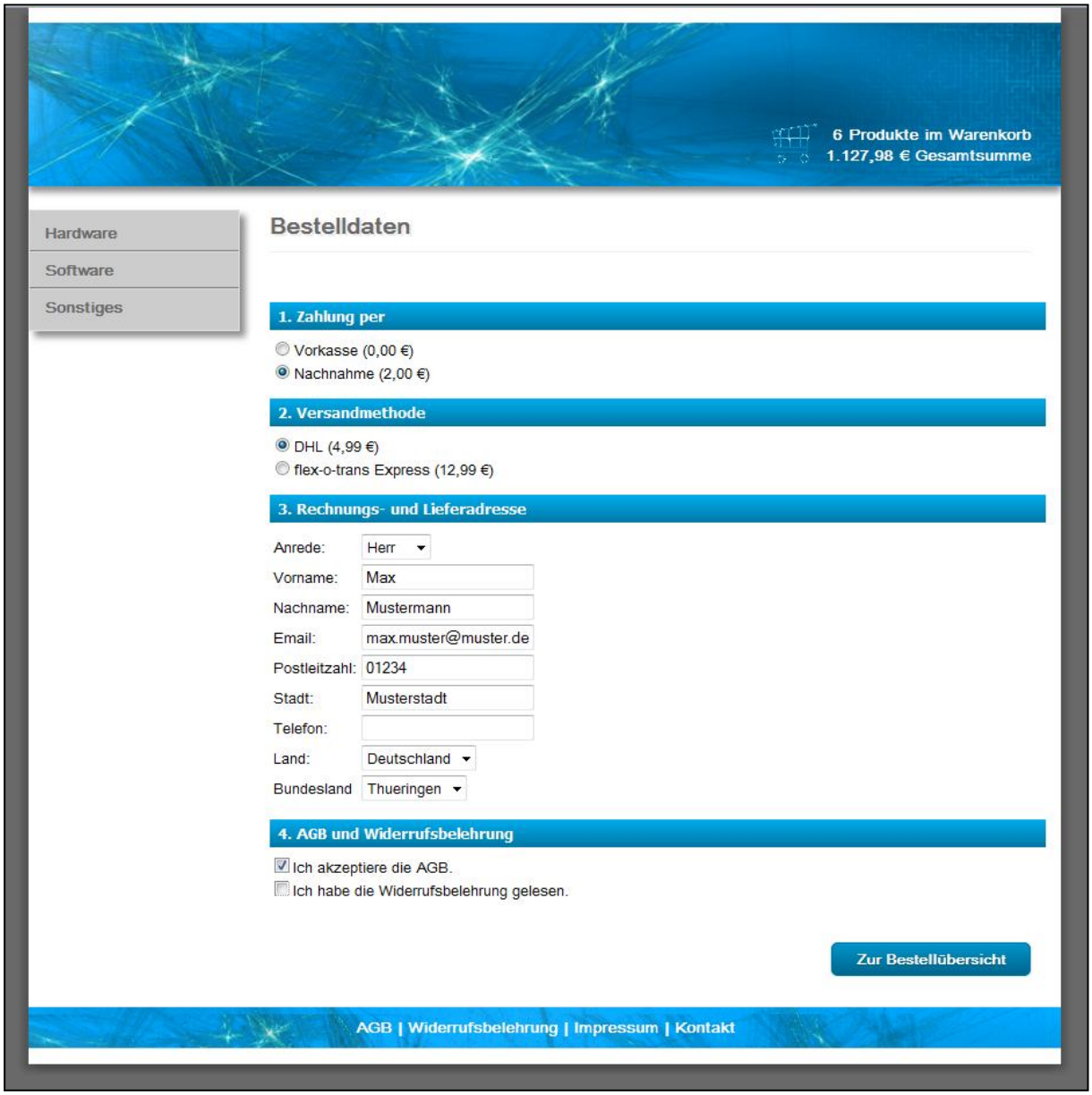
**Warenkorb**



## Anlage 4: Front-End-Bilder

### Blatt 3

**Formular - Bestelldaten**



**Erklärung**

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

| | |
|---|---|
| Bearbeitungsort, Datum | Unterschrift |